

A Flexibility Architecture.

Dr. Tom Johnston
Mindful Data, Inc.
Written sometime in 2001.

Note: now (June, 2008). The extreme flexibility described in this short essay would be most valuable to an application software vendor. It would permit them to sell a single codebase and set of database schemas, and be able to accommodate the specific requirements of each customer without altering that codebase or those schemas, and to develop and distribute new releases without altering the codebase or schemas. It is, in fact, the recipe for a "universal software application".

If "total cost of ownership" were ever taken seriously enough to be measured over the lifetime of a business application, it would be immediately apparent that much of this architecture would be cost-justified many times over for in-house applications as well.

The basic architecture which I propose will support a declarative approach to business software flexibility, consists of five layers.

The Core Layer.

First, there is a core layer, consisting of a highly generic database schema (tables, attributes, relationships), and procedural code and queries that understand that core schema. At this point, the database schema is so generic that it gives no indication what the actual business application is.

What "understand" means, in this context, is that the core code and queries are compile-time bound to the core schema. But here's where the flexibility comes in: they are compile-time bound to *nothing more than the core schema*. Because they aren't written with an awareness of any data structures other than the core schema structures, they will work with interpretations and extensions of those core structures, without modification.

Flexible Architecture Layers	Database	Codebase
The core layer.	The core schema: a highly generic physical database schema.	The core code: procedural code and queries which manipulate the core schema.
The specification layer.	Metadata tables.	Specification management code: code and a user interface (UI) to manage the metadata.
The interpretation layer.		Interpretation code: code which uses the metadata to translate between the core schema and the business views of the data.
The business layer.	Business views of the database.	Business logic: the core code as it uses run-time metadata to manipulate business-specific views of the database.
The presentation layer.	Highly generic UIs—screens and reports.	Presentation logic: code to present the business objects to the user, and translate his commands to the system.

Figure 1. Layers of the Flexible Architecture.

The Specification Layer.

The second layer is a specification layer. It is the administrative component of the business application. Through its screens, the system administrator defines the business rules which specify the business application. These business rules are stored in metadata tables. The rules describe what kinds of things (entities, object classes) the application manages, how many attributes each entity or class has and what those attributes are like, and what relationships each entity or class has, and what those relationships are like.

The Interpretation Layer.

The third layer is an interpretation layer. At run-time, the metadata tables are read and interpreted, and this processes “fleshes out” the core schemas. For example, one day a business system might permit an employee to report to only one manager, and the next day it might permit an employee to report to multiple managers. The “business as usual” way of implementing this change would be to alter the database by creating a new associative table, change all affected code and queries, change screens and reports, and transform and re-load existing data—a process that typically would take months. The declarative way of implementing this change is to sit down at a screen, and change one row of one metadata table—a process that typically would take a minute or so. No changes to the database schemas; no changes to code; no transformation and re-loading of data.

The Business Layer.

The fourth layer is the business logic itself. In traditional applications, this is simply hardcoded, and the first three layers of this architecture do not exist. In this architecture, however, what is hardcoded is code capable of using metadata to manipulate business objects which are fully defined only after that code is written and compiled. All the metadata in the world won’t do much good unless the base code is intelligent enough to read it, interpret it, and act on it.

This kind of code effectively modifies itself at run-time, by reading and interpreting the metadata. For example, the code might examine a table of valid values for an attribute, in order to validate an entry for it. If a new value is added, then the next time the code is executed, instances can be created which contain that new value for that attribute. To take another example, the code is not compiled with a knowledge of how many attributes characterize each entity or object class. Instead, it is written and compiled to manage the attributes of any entity or class as a variable-length list. And it finds out how long this list is (how many attributes there are for a particular entity or class) and the data type and length of each attribute in the list, only at run-time.

The Presentation Layer.

The fifth layer is a new kind of screen and report design. Categories of things are displayed on screens through a Microsoft Explorer-like window, usually on the left side of the screen. This lets us navigate a tree-structure of types of things—entities or object classes. The structure may be a supertype/subtype structure, *e.g.* a class inheritance structure. Or it may be an aggregation structure, *i.e.* a structure of containers and the things they contain. Specific instances of things are displayed, one-by-one, as extendible lists of attribute names and values. Relationships among specific instances of things are displayed as lists, each of whose members consists of the name of the type of relationship, and an identifier and name of the related thing.

The advantage of this kind of UI is that its compiled form does not need to change if new entities or classes are added, just as MS Explorer doesn't need to change when new file folders are added. It does not need to change when new attributes are added to an entity or class, because it displays attributes as a scrollable list, and discovers the length of the list, and the names of the attributes in the list, at run-time.