MANAGING THE EARTHQUAKE: SURVIVING MAJOR DATABASE ARCHITECTURE CHANGES

Michael Rosenblum, Dulcian Inc

INTRODUCTION

Changes are inevitable in any information system. Even with the best possible requirements analysis, it is not possible to anticipate all of the possible changes that may be required in a system over time. The best solution is to try to predict as many potential failure points as possible in order to reduce the chances of such failures to an "acceptable" level.

Used in this sense, "acceptable" is greatly dependent upon the project budget and management tolerance. However, in any efficiently run organization, proposing system modification costs that are comparable to the original system development expenditures could be considered a “fireable offence.” The idea is to build systems in such a way that the inevitable changes are less likely to make the system fail.

REAL WORLD EXAMPLES

As an example of bad system design, the following problem was faced by our development team while enhancing and rebuilding the Budget and Finance System for the Finance Ministry of Ethiopia. In the existing data warehouse, a decision had been made to use “smart codes” for organization IDs so for each Region/Zone/City, etc., each block had a fixed length. Since all values were fixed-length, the simplest way to check the type of organization was to check the length of the ID (Ex. 14 means city, 20 means zone, etc).

Unfortunately, at some point an extra level was introduced for reporting purposes, which destroyed almost 100% of the existing organization-based reports. The result required enormous effort to repair and devise a workaround to glue the system back together.

On the positive side, using best practices when building a system for the United States Air Force Reserve, it was possible to handle some significant changes without expending a great deal of time money. The following were achieved in very little time with minimal disruption to the existing production system:

- Shift from single to multi-organization structure
  - This involved adding users from Air Force Active Duty and Air National Guard to the Reserve system and introducing the notion of shared vs. organization-specific data
  - Development time: 3 months
- Adding an extra level of organization hierarchy across the board
  - Development time: 1 month
- Introducing offline capabilities for a module with approximately 300,000 lines of generated PL/SQL
  - Development time: 3 months
- Maintenance of a local XML database with a single Java-based generator (1000 lines of code)

PROPOSED STRATEGIES FOR SUCCESS

Fifteen years of experiences at Dulcian have led to the development of the following three most efficient empirical strategies:

1. Robust data models are less likely to be disrupted by changes.
2. Good system architecture may allow major changes to be made inexpensively.
3. A "thick database" approach is less susceptible to inevitable UI architecture shifts.

Although this list is neither exhaustive, nor carved in stone, it has proven effective in many situations and provided the highest ROI for an organization's IT budget.

DATA MODELS

In the current database development environment, the trend of pushing everything into the middle-tier and using the database as nothing more than persistent data storage, the skill of effective data modeling becomes lost mainly due to the shifting of complexity to different areas. However, there are some common modeling patterns and anti-patterns that persist:

1. Patterns:
    a. Type instead of multiple associations
    b. Org Unit instead of specifically designated types
    c. Generic tree structures
2. Anti-patterns:
    a. Hard-coded structures
    b. Smart attributes
    c. Over-generalizing

The following sections describe several real world examples of these patterns in use in production systems.

A. TYPES VS. HARD-CODED STRUCTURES

At Dulcian, an official system development policy was set up whereby the request to add a second association of the same kind between two classes MUST include a guarantee that there will NEVER be a third association. If this guarantee is not possible, then an intersection class is created.

This policy arose when building a system for the US Coast Guard. The original specifications defined that ships were repaired by special organizations. In later stages of the project, it was discovered that organizations could also manage and maintain ships. Since the deadline for completion was imminent, the in-house development team unanimously voted to add two more associations between SHIP and ORG, so there would be no impact on the existing code. Unfortunately, there was no guarantee that there would be no more surprises later. It took a direct act of management to implement the generic structure shown in Figure 1.

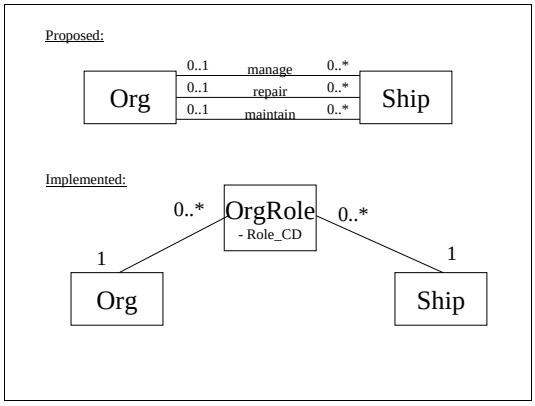

Figure 1: Generic Organization Model Structure

Two years later, a system audit revealed seven different types of roles in ORGROLE table, not just the original three. Introducing each of these additional types took significantly less time and required less money than anticipated.

Lesson to be learned: if it is not ONE, it is MANY.

B. Organizations and Organization Trees

Another common best-practice used at Dulcian is building an organization structure as an abstract tree of elements, where each element in the tree is an organization unit of a specific type as shown in Figure 2.

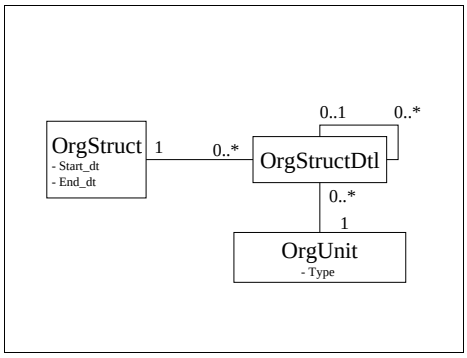

Figure 2: Abstract Tree of Elements

This concept has proven its effectiveness a number of times:

1. Adding an extra organizational level required minimal development instead of a major system rewrite

2. Creating a common structure for other armed services branches became possible by adding 10 extra organization types and providing rules about how to correctly link them together
3. Time-stamping the whole tree (rather than separate elements) made reporting straightforward, because at any point in time there could have been one, and only one, active tree for reports to be based upon.

After several years when the number of nodes reached the thousands, some “nice-to-haves" were introduced in order to make development easier and to maximize system performance:

1. “Current” snapshots are used for day-to-day activities. This strategy avoids expensive hierarchical searches. In addition, materialized views are automatically refreshed every night. The snapshots have the following characteristics:
    a. They represent the current state of the tree.
    b. They contain a lot of indexes.
    c. They are optimized for querying (using a lot of denormalization).
2. History lookup tables are used for reporting purposes. They “flip” the data. Instead of answering the question "What organization belongs to what tree?", they show the length of time that the organization belonged to the specified chain of command. These tables are appended when a new organization tree becomes active and they are heavily denormalized.
3. A “time machine” manages scheduled changes without forcing them to coexist with the current data. This structure:
    a. is represented as a special change log table
    b. applies changes nightly
    c. allows detection of mutually exclusive future changes and nulls them out

C. Over-generalizing (anti-pattern!)

It is a common trap for many new developers when discovering the power of generic data models to "over-abstract" the data model and end up with something like the "Stuff" model shown in Figure 3.

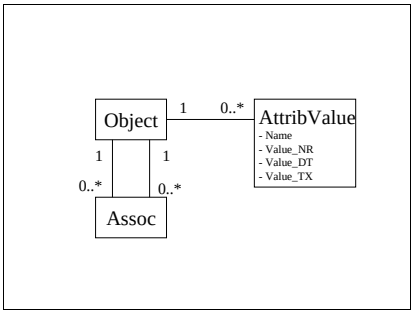

Figure 3: "Stuff" model

Although this model articulates everything and works just fine for SCOTT/TIGER-level problems, (i.e. very localized data models with few tables). However, trying to scale these models for large production systems will cause them to collapse under their own weight, because the cost of retrieving of a single data element from this type of structure is comparable to hard-coded solutions. In other words, beware!

System Architecture

Very often the best way of avoiding future problems when building a system is to radically change the way in which to approach the problem. This may not only solve the original problem much more efficiently, but also lead to other improvements as illustrated by the following examples.

Creating a Data Mapper

The core of the system being built involved maintaining hundreds of different forms. Prototypes of all forms were hardcoded, which caused major development issues any time additional rules or data structures needed to be added to the system. Abstracting the process helped to solve this problem using the following elements:

- Generic repository to define object transformation
- Data in the database → data in the form
- Data in the form → data in the database
- Code Generator (for performance reasons)

Comment [CLF1]: It is not clear what the arrows represent – please replace with words.

The model shown in Figure 4 is based on the following points:

1. All required structural objects and their attributes are defined in the database.
2. A map is a process that reads from the source object and writes into the destination object.
3. An original data element from the source can be transformed via a specified expression (map_column)

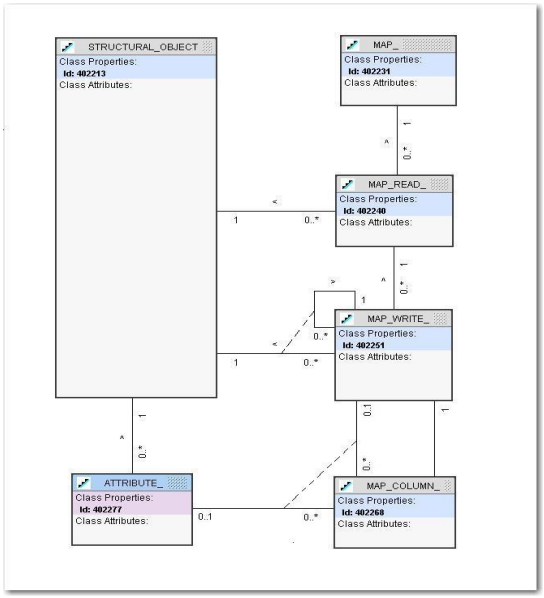

Figure 4: Mapper Data Model

Although the process of changing the data model required several months, once the shift was complete, the development process was greatly accelerated. Two hundred additional forms were added in two months (instead of the originally expected 12 months); bug fixes could be made in a matter of minutes; and major data model changes only required a few days to update the dependencies in the repository.

In addition to the development speed benefits, the same repository could support much more including:

- Data transformation inside the database
- Data migration
- Processing data to/from XML
- Offline data export

Currently, there are more than 300,000 lines of PL/SQL code generated from the same repository which demonstrated that creating a single generator is much more efficient than writing hundreds of hard-coded routines.

EVENT-ACTION FRAMEWORK

One project implemented in a developing nation required a major shift in system architecture. The biggest challenge was the network connection speed (often less than 5K). In order to deploy the solution over the web, there were two problems to solve:

1. Minimize the number of requests between the client and application server
2. Keep the page size as small as possible.

Both of these goals were achievable using the event-action framework shown in Figure 5.

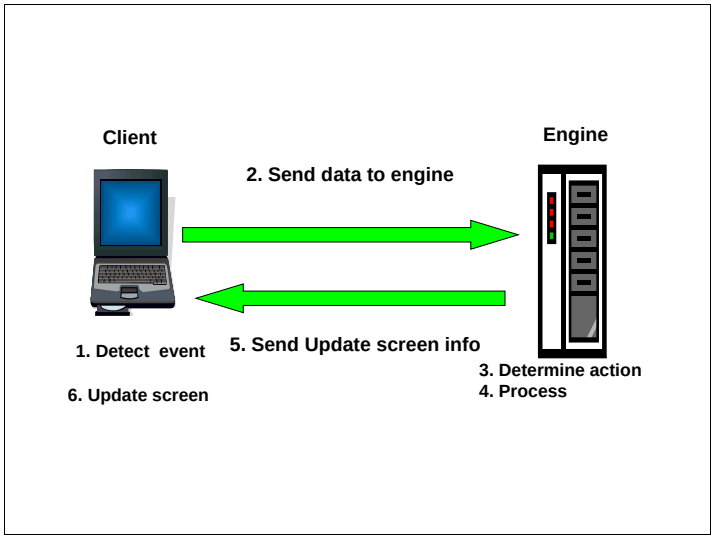

Figure 5: Event-Action Framework

Instead of transferring complete (or even partial) pages between the engine and the client, the client only indicates any changes to the engine (event) and the engine performs the appropriate actions and applies them to the known state of

the client. Since the server is aware of the old (before action) and new (after action) client states, the only information required deals with any visible changes. For this system, the average transfer was less than 1K per screen.

"THICK DATABASE" APPROACH

This concept is still considered by many to be "heresy" in the IT world. However, based on our experiences, following these three "thick database" approach best-practices significantly increases the chances of project success:

1. The user interface screens never touch a table. Different denormalized views are utilized.
2. All bulk operations are done in PL/SQL.
3. UI logic is pushed into the database as much as possible.
    a. Validation – creating reusable rules improves their overall visibility
    b. Page flow – "Traffic cop" implemented as a set of PL/SQL models makes the application logic manageable.
    c. Even partial setting of the user interface layout from the database significantly increases system flexibility.

This section will focus on the first rule since the other are outside of the scope of pure database development. The idea is to convert relational data into something that will make user interface development easier. The best way to accomplish this is to separate the data representation in the front-end from the model. The solution is to specify all columns from all tables (with appropriate joins) that would be needed and create a view as shown here:

```
create or replace view v_customer
as
select c.cust_id,
       c.name_tx,
       a.addr_id,
       a.street_tx,
       a.state_cd,
       a.postal_cd
from customer c
left outer join address a
   on c.cust_id = a.cust_id
```

To make this view updatable, a special kind of trigger (INSTEAD OF) can be created. The INSTEAD OF code construction can be used to Insert, Update and Delete as shown in the following three code samples:

```
create or replace trigger v_customer_ii
instead of insert on v_customer
declare
  v_cust_id customer.cust_id%rowtype;
begin
  if :new.name_tx is not null then
   insert into customer (cust_id,name_tx)
    values(object_seq.nextval,      :new.name_tx)
   returning cust_id into v_cust_id;
  if :new.street_tx is not null then
   insert into address (addr_id,street_tx,
        state_cd, postal_cd, cust_id)
   values (object_seq.nextval,:new.street_tx,
    :new.state_cd,:new.postal_cd, v_cust_id);
  end if;
end;
```

```
create or replace trigger v_customer_id
instead of delete on v_customer
begin
    delete from address
    where cust_id=:old.cust_id;
    delete from customer
    where cust_id=:old.cust_id;
end;


create or replace trigger v_customer_iu
instead of update on v_customer
begin
 update customer set name_tx  = :new.name_tx
 where cust_id = :old.cust_id;

 if :old.addr_id is not null
 and :new.street_tx is null then
  delete from address where addr_id=:old.addr_id;
 elsif :old.addr_id is null
 and :new.street_tx is not null then
  insert into address(…) values (…);
 else
  update address set… where addr_id=:old.addr_id;
 end if;
end;
```

It is not always possible to represent all required functionality in a single SQL statement, which means that denormalized view cannot be built. But Oracle has provided a different mechanism, namely collections, that allow you to hide not only the data separation, but also all of the transformation logic.

Nested tables can be used in SQL queries with the special operator TABLE. This allows you to hide the complex procedural logic "under the hood." The nested table type must be declared as a user-defined type (CREATE OR REPLACE TYPE...) It is also necessary to specify exactly what is needed as output and declare the corresponding collection as shown here:

```
Create type lov_oty is object (id_nr NUMBER,
                               display_tx VARCHAR2(256));
Create type lov_nt as table of lov_oty;
```

To write a PL/SQL function that hides all required logic, you can use code as shown here:

```
function f_getLov_nt
 (i_table_tx,i_id_tx,i_display_tx,i_order_tx)
return lov_nt is
  v_out_nt lov_nt := lov_nt();
begin
  execute immediate
    'select lov_oty('
          ||i_id_tx||','||i_display_tx||
                ')'||
    ' from '||i_table_tx||
    ' order by '||i_order_tx
  bulk collect into v_out_nt;
  return v_out_nt;
end;
```

There is now a function that can return an object collection. An object collection can be cast to a table. Object collections types are supported in SQL. Therefore, a function call could be used as a data source in SQL. You can test it using the following code:

```
select id_nr, display_tx
from table(
          cast(f_getLov_nt
                 ('emp',
                  'empno',
                  'ename||''-''||job',
                  'ename')
          as lov_nt)
          )
```

Finally, create a view on top of the SQL statement. This completely hides the underlying logic from the UI. INSTEAD-OF triggers make the logic bi-directional. One minor problem is that there is still no way of passing parameters into the view other than by using some type of global as shown here:

```
Create or replace view v_generic_lov as
select id_nr, display_tx
from table( cast(f_getLov_nt
            (GV_pkg.f_getCurTable,
             GV_pkg.f_getPK(GV_pkg.f_getCurTable),
             GV_pkg.f_getDSP(GV_pkg.f_getCurTable),
             GV_pkg.f_getSORT(GV_pkg.f_getCurTable))
            as lov_nt)
          )
```

Comment [CLF2]: You need some text here – Don't end a section with code.

SUMMARY

There are many different approaches to help make systems more reliable and sustainable over time that could be discussed at length. To summarize the most important ones:

- Good system architecture does matter.
- Good system architects play a key role in building successful systems.
- Devoting adequate time and resources to good system architecture in the short run will always pay off in the long run.

ABOUT THE AUTHOR

Michael Rosenblum is a Development DBA at Dulcian, Inc. He is responsible for system tuning and application architecture. He supports Dulcian developers by writing complex PL/SQL routines and researching new features. Mr. Rosenblum is the co-author of *PL/SQL for Dummies* (Wiley Press, 2006). Michael is a frequent presenter at various regional and national Oracle user group conferences. In his native Ukraine, he received the scholarship of the President of Ukraine, a Masters Degree in Information Systems, and a Diploma with Honors from the Kiev National University of Economics.