Data-driven Programming

By

Johan Nel

A series of articles explaining the principles

Article 6: EAV and OLT: Enhancing the concepts

November 2014

Table of contents

1. INTRODUCTION 1

2. A PERSISTENT INTERFACE LAYER 1

3. ONE (TRUE) LOOKUP TABLE (OTLT) 1

3.1 THE BASIC LOOKUP TABLE 2
3.2 THE "BETTER/PERSISTENT" LOOKUP TABLE DESIGN 3
3.3 OTLT 4
3.4 TWO TABLE LOOKUP (TTL) 6

4. ENTITY ATTRIBUTE VALUE (EAV) 7

5. CHANGING OUR DATASTORE TO TTLT & EAV 9

6. SUMMARY 15

Tables

TABLE 1: BASIC LOOKUP TABLE STRUCTURE 2
TABLE 2: BASIC LOOKUP STRUCTURE 3
TABLE 3: ONE LOOKUP STRUCTURE 4
TABLE 4: ONE LOOKUP TABLE WITH GENDER AND HAIRCOLOUR 5
TABLE 5: ONE LOOKUP TABLE COMPROMISE 6
TABLE 6: TWO TABLE LOOKUP MASTER 6
TABLE 7: TWO TABLE LOOKUP DETAILS 6
TABLE 8: DATABASE CLASS TABLE STRUCTURE 14
TABLE 9: DATABASE CLASS TABLE STRUCTURE MODIFIED FOR EAV 14

Listings

LISTING 1: TYPICAL LOOKUP DATABASE SCRIPT AND TABLE REFERENCING IT 2
LISTING 2: INTERNAL UNIQUE ID LOOKUP TABLE IMPLEMENTATION 3
LISTING 3: ONE LOOKUP TABLE IMPLEMENTATION 5
LISTING 4: ONE LOOKUP TABLE REDESIGN IMPLICATIONS 5
LISTING 5: TWO TABLE LOOKUP IMPLEMENTATION 7
LISTING 6: CHECKING FOR ENTEGRITY ISSUES WITH A TWO TABLE LOOKUP APPROACH 7
LISTING 7: ARTICLE 4 DATASTORE DEFINITION 9
LISTING 8: DATA STORE BREAKDOWN INTO SMALLER COMPONENTS 10
LISTING 9: DATA STORE CLASS TABLE 11
LISTING 10: TTLT LKPDEF & LKPITEM 11
LISTING 11: DATA STORE CLASS TABLE REFERENCING THE TTLT (CLASS.CLASSTYPE_NO REF LKPITEM.LKPITEM.LKPITEM_NO 11
LISTING 12: DATA STORE CLASS MEMBER TABLE REFERENCING THE TTLT 12
LISTING 13: CLASS AND MEMBER VIEWS INTERFACING TO TTLT 12
LISTING 14: DATA STORE FORMAT 13
LISTING 15: DATA STORE WITH SPARSELY FILLED PROPERTIES FORMAT 14

1. Introduction

In the previous article we have created our first working data-driven application doing exactly what it needs to do, without actually knowing what it is doing, or for that matter what it will be doing in the future☺ However, all it knows is that it has a [Application]Form and at the moment it can contain some members. It knows where to (the MemberInterface) ask for members. Our interface layer is a bit limited though, it only knows how to communicate (delegates) and can only fetch MenuStrips and MenuStrip Items and tell them how to become members via delegation. We will however take a bit of an off the beaten track approach with this article. Well I know you might think we already off the track with a data-driven approach, but we will look at ways how we can make our interface layer persistent.

2. A persistent interface layer

If we look at our datastore (<Application>.exe.ini), it is easy to see that just as in the case of business applications our types of containers in our store will grow as we add new features. A customer table will soon have an order table, which in turn will have an order item table. We need to also add to our presentation layer some process to communicate with them. It is the same with data-driven applications. However, that is one of the fundamental issues that I believe why developers shy away form the concept. They do not think abstract enough. We will therefore look at ways of trying to achieve a persistent datastore and a persistent internal interface for know and unknown members. Hence the topic of this article OLT & EAV.

3. One (True) Lookup Table (OTLT)

During software and database development, we encounter frequently many tables consisting of Identity and Value pairs. I will not go into the details, but there were theorists who say why do we need all these similar little small tables. Lets rather create one big Identity Value table and soon the concept of OTLT was created. There are however positives and negatives. The purists immediately shoot it down, saying that we create another maintenance nightmare to ensure the property (column or field) will only allow the correct values. It is a true statement if not implemented correctly. It does however reduce the data base administrator's task of creating all these repetitive tables. However, to ensure integrity, it burdens the administrator and or developer's task of creating integrity rules. In most cases what will

happen is that it is developer driven, and they will quickly from their perspective indicate that the integrity rules should be application driven. The application will ensure that the integrity is applied. Unfortunately, what happens if somebody not knowing (the new DBA) adds items to the OTLT directly via the database bypassing the system integrity rules? We can easily end up with a sexual orientation column (Hetero, Homo, Bi) saying Sheep which should actually be part of a selection for farming (no pun intended)☺. Well that was a bit tongue in the cheek, however if you look at the doom prophets of the concept and the examples they provide of what can go wrong that might be the case. My rule of thumb is that it does have its merits when we only need an identifier and a description. When more than a description is however required to describe the feature, create a separate table. In most cases though, when we working with abstract programming it is a bit of a different ball game…

In this section we will look at the concept and how I believe it can be utilised effectively without creating potential anarchy.

3.1 The basic lookup table

Lets start with the basic structure of a lookup table (Table 1) and we will look at a gender column in our database describing a person containing a property (column) gender. We will have a table gender that is referenced from person and we will probably create an integrity check to indicate it is required and that our gender table records can only be deleted if there is no person with the gender type that became obsolete (Listing 1):

Table 1: Basic lookup table structure

Gender_ID	Gender_Description
M	Male
F	Female
U	Unknown

Listing 1: Typical lookup database script and table referencing it

```
CREATE TABLE gender
 gender_id VARCHAR(1) NOT NULL PRIMARY KEY,
 gender_description VARCHAR(10) NOT NULL;

INSERT INTO gender VALUES ('F', 'Female'), ('M', 'Male'), ('U', 'Unknown');

ALTER TABLE person
 ADD gender_id VARCHAR(1) DEFAULT 'U' NOT NULL REFERENCE gender(gender_id)
            ON DELETE RESTRICT ON UPDATE CASCADE;

CREATE VIEW personview AS
 SELECT p.*, g.gender_description FROM person p, gender g WHERE p.gender_id = g.gender_id;
```

All well, miraculously after running the above sql in our database admin tool all persons will have a gender type ‘U’ and we can start correcting the column till we don’t have any more Unknown genders where we know thegender of each person.

Looking at the database script we can state that it seems to be fair and will cater for all that is required. We can insert new gender types and update person to consume the new gender type. We can delete gender types and our database will ensure we don’t delete any if the gender type if it is still in use. Mission accomplished. Oh, we missed one! We can update or code for female to rather use ‘W’:’Woman’ and all persons with gender ‘F’ will change to ‘W’. Warning bells however, if we change ‘U’:’Unknown’ to ‘N’:’Not known’, suddenly we have troubles when we insert a new person and do not specify gender☹ Our person table need some care. We want to make our database persistent. An anomaly can creep in.

3.2 The “better/persistent” lookup table design

Our gender lookup table in section 3.1 has a potential for failing, we cannot change the code used for Unknown gender and hope our database will still run happily after. A principle I implement in practise is that I try to eliminate cascaded updates as far as possible at database design time. Ensuring persistence, I rather make use of a lets call it “internal unique id” and a “business unique id”. There are different ways to implement this, some might prefer to use a GUID, I prefer to use a Sequence (AutoNumber, Serial) type. Different RDBMS’s use diffirent names for this feature. I will make use of my preferred RDBMS (PostgreSQL) syntax of [big]serial. Table 2 shows the typical table and Listing 2 the script changes required in blue.

Table 2: Basic lookup structure

Gender_No	Gender_ID	Gender_Description
1	U	Unknown
2	M	Male
3	F	Female

Listing 2: Internal unique id lookup table implementation

```
CREATE TABLE gender
 gender_no SERIAL NOT NULL PRIMARY KEY,
 gender_id VARCHAR(1) NOT NULL UNIQUE,
 gender_description VARCHAR(10) NOT NULL;

INSERT INTO gender
       (gender_id, gender_descripton)
VALUES (‘U’, ‘Unknown’), (‘M’, ‘Male’), (‘F’, ‘Female’);

ALTER TABLE person
```

```
  ADD gender_no INTEGER DEFAULT 1 NOT NULL REFERENCE gender(gender_no)
            ON DELETE RESTRICT ON UPDATE CASCADE;

CREATE VIEW personview AS
 SELECT p.*, g.gender_id, g.gender_description
  FROM person p, gender g WHERE p.gender_no = g.gender_no;
```

Again, after running the above, our person table will have a default gender[_no] = 1. When viewing through our personview, it will show that gender[_no] = 1, is associated with gender_id = 'U', with gender_description = 'Unknown'. We can add new gender types, we can update the business unique id, and our integrity will be intact. What we need to ensure is that we always add the default first into our lookup table hence having a serial value of 1.

There are many practises used for naming conventions. My preference (up for debate) is to re-use the lookup table unique identifier, that way I do not have to remember that person.gender = gender.gender_no, it can be altered as preferred by everyone, just be persistent or is it consistent. I always use <gender>_no for references and persistently name all columns <gender>_no. Nobody needs to try and figure out that gender refers to gender_no in other tables. <Gender> will also be the name of the table where it is (normally) maintained in. In any other table it will be a reference. When I need to define a number as column in a table, e.g phone number, I use the practise of using <phone>_nr. It is easily identified that it is a property that the user can manipulate directly.

You probably think, so what does this have to do with OTLT? More in the next subsection.

3.3 OTLT

According to the one lookup table concept we don't need a gender lookup table, the reason be that we can also have a hair colour property describing a person. Lets start creating our one lookup table (Table 3), and we fill it with our gender details that we know.

Table 3: One lookup structure

LkpItem_No	LkpItem_ID	LkpItem_Description
1	U	Unknown
2	M	Male
3	F	Female

We will now update our person table (Listing 3).

Listing 3: One lookup table implementation

```
CREATE TABLE lkpitem
 lkpitem_no SERIAL NOT NULL PRIMARY KEY,
 lkpitem_id VARCHAR(1) NOT NULL UNIQUE,
 lkpitem_description VARCHAR(10) NOT NULL;

INSERT INTO lkpitem
       (lkpitem_id, lkpitem_descripton)
VALUES ('U', 'Unknown'), ('M', 'Male'), ('F', 'Female');

ALTER TABLE person
 ADD gender_no INTEGER DEFAULT 1 NOT NULL REFERENCE lkpitem(lkpitem_no)
            ON DELETE RESTRICT ON UPDATE CASCADE;

CREATE VIEW personview AS
 SELECT p.*, l.lkpitem_id as gender_id, l.lkpitem_description as gender_description
  FROM person p, lkpitem l WHERE p.gender_no = l.lkpitem_no;
```

We can give self a pat on the shoulder, we have just created a platform for less work☺ All we need to do now is add out haircolours (Table 4) and adapt our person table accordingly (Listing 4).

Table 4: One lookup table with gender and haircolour

LkpItem_No	LkpItem_ID	LkpItem_Description
1	U	Unknown
2	M	Male
3	F	Female
4	B	Black
5	R	Brown
6	E	Red
7	L	Blond

Listing 4: One lookup table redesign implications

```
INSERT INTO lkpitem
       (lkpitem_id, lkpitem_descripton)
VALUES ('B', 'Black'), ('R', 'Brown'), ('E', 'Red'), ('L', 'Blond');

ALTER TABLE person
 ALTER gender_no CHECK gender_no = 1 OR gender_no BETWEEN 2 AND 3
 ADD haircolor_no INTEGER DEFAULT 1 NOT NULL REFERENCE lkpitem(lkpitem_no)
            ON DELETE RESTRICT ON UPDATE CASCADE
            CHECK haircolour_no = 1 OR haircolour_no BETWEEN 4 AND 7);

CREATE VIEW personview AS
 SELECT p.*,
        g.lkpitem_id as gender_id, g.lkpitem_description as gender_description,
        h.lkpitem_id as haircolor_id, h.lkpitem_description as haircolour_description
  FROM person p, lkpitem g, lkpitem h
  WHERE p.gender_no = g.lkpitem_no and p.haircolour_no = h.lkpitem_no;
```

Our database will keep integrity based on what we know. However, what if we need to add another haircolour? Not too much trouble we can add ('8','Ashblond') and update person to allow a range of 4 to 8. We will have a problem though if we discover a new gender type, our lookup table work on the principle of grouped items. We do not cater for disjointed groups. If you look at the purists shooting down the concept this is how they approach it. What we

need, due to creating less tables, is our columns in the table needs to expand. Gain by less tables, compromise with an additional column(s) (Table 5). I also add another column called default.

Table 5: One lookup table compromise

LkpItem_No (PK)	LkpColumn_ID (UK1), (UK2)	LkpItem_ID (UK1)	LkpItem_Description	Default (UK2)
1	Gender_no	U	Unknown	Y
2	Gender_no	M	Male	N
3	Gender_no	F	Female	N
4	Haircolour_no	B	Black	N
5	Haircolor_no	R	Brown	N
6	Haircolour_no	E	Red	N
7	Haircolour_no	L	Blond	N
8	Haircolour_no	U	Unknown	Y

I hope this make sense. There is one issue in the above, and maybe you observed it, but it does not matter. The Brown hair colo(u)r will get lost in our database. A typing aka human error. It is therefore that I rather prefer a Two Table Lookup approach.

3.4 Two Table Lookup (TTL)

Well as you guess this approach is a bit more work, however we achieve better integrity. Table 6 and Table 7 list the table structure:

Table 6: Two table lookup master

LkpDef_No (PK)	LkpDef_ID (UK)	LkpDef_Description
1	gender_no	Gender of a person
2	haircolour_no	Haircolour of a person

Table 7: Two table lookup details

LkpItem_No (PK)	LkpDef_No (UK1), (UK2)	LkpItem_ID (UK1)	LkpItem_Description	Default (UK2)
1	1	U	Unknown	Y
2	1	M	Male	N
3	1	F	Female	N
4	2	B	Black	N
5	2	R	Brown	N
6	2	E	Red	N
7	2	L	Blond	N
8	2	U	Unknown	Y

We need to alter our table logic to incorporate this two table lookup concept.

Listing 5: Two table lookup implementation

```
CREATE TABLE person
 …
 gender_no INTEGER NOT NULL REFERENCE lkpitem(lkpitem_no),
 haircolour_no INTEGER NOT NULL REFERENCE lkpitem(lkpitem_no)
            ON DELETE RESTRICT ON UPDATE CASCADE
            TRIGGER ON INSERT OR UPDATE person_trigger_b_ins_upd;

CREATE TRIGGER FUNCTION person_trigger_b_ins_upd AS
  IF NEW.gender_no IS NULL
    SELECT lkpitem_no FROM lkpdef d, lkpitem i
     WHERE d.lkpdef_no = i.lkpdef_no
       AND d.lkpdef_id = 'gender_no'
       AND lkpitem.default = true INTO NEW.person.gender_no
  ELSE
    SELECT lkpdef.lkpdef_id FROM lkpdef d, lkpitem i
     WHERE d.lkpdef_no = i.lkpdef_no
       AND i.lkpitem_no = NEW.gender_no INTO col_name;
   IF col_name <> 'gender_no'
     THROW 'Gender does not contain a valid reference'
   ENDIF;
  IF NEW.haircolour_no IS NULL
   --Repeat the above, same as gender_no check. Just replace gender with haircolour
  ENDIF;
RETURN TRIGGER;

CREATE VIEW personview AS
 SELECT p.*,
        g.lkpitem_id as gender_id, g.lkpitem_description as gender_description,
        h.lkpitem_id as haircolor_id, h.lkpitem_description as haircolour_description
  FROM person p, lkpitem g, lkpitem h
  WHERE p.gender_no = g.lkpitem_no and p.haircolour_no = h.lkpitem_no;
```

We have successfully created a Two Lookup Table approach that is both manageable, but also ensure integrity at database level.

And for checking if something has slipped through the cracks, we can do the query per Listing 6 and replace 'gender_no' with the applicable column we want to verify.

Listing 6: Checking for entegrity issues with a two table lookup approach

```
SELECT * FROM person p, lkpdef d, lkpitem i
 WHERE p.gender_no = i.lkpitem_no
  AND i.lkpdef_no = d.lkpdef_no
  AND d.lkpdef_id <> 'gender_no';
```

Enough said about the concept of lookup table design, it is time to look at Entity Attribute Value.

4. Entity Attribute Value (EAV)

According to references (Google & Wikipedia and other sources) the concept of EAV originated from clinical data. In general it implies that not all data is applicable and we create a sparsely filled table where most of the columns will have no data, or we have a rapid increase in new data requirements. Our poor DBA is thrown in over his head, and we have two options, we appoint another DBA to spread the workload, or we work smarter. The idea

of EAV is that we know some basic data requirements, e.g. in the clinical environment we have patients. We will define a unique identity for the patient: social security number, ID document number, passport number, etc. and some descriptive data: firstname, lastname, title, address, etc. However we need now to define illness. There are quite a number of illnesses the patient can have: from none, to quite a few. It is almost impossible to define a table that can contain a column for all potential illnesses. We also have a problem in presenting it to the application user, since the screen manufacturers think it is crazy that we need a 20ft x 15ft screen to ensure our illness form is fitting onto 1 screen. Well maybe not crazy, but at minimum we smoking or sniffing stuff. Another field where this is a scenario is in the research arena. They start out with a basic list of data required, and as the data is analysed, new data requirements pop up.

So how does the concept help with managing data?

Lets look at the basic design for a research trial and we define that we need to have a research identification, the purpose and data for data1, data2, data3 and data4 measured over a period. Off we go and design our database:

Trial_ID	**Purpose**
0001/001/00001	Trial to investigate the inter-dependency between data1 to data4 over time since application of fertiliser XYZ

Trial_ID	**Measurement_Date**	**Data1**	**Data2**	**Data3**	**Data4**
0001/001/00001	2012-01-15	0.1	0.3	0.7	0.9
0001/001/00001	2012-01-16	0.3	0.1	0.6	0.0

After the first couple of weeks into the trial it is realised that Data4 needs a further breakdown, which leads to researchers believing that this new data4.8 potentially have an interaction to Data1 after statistical analysis of a couple of weeks data gathering.

And here is where the EAV model comes into play.

We rather develop a 3 table approach where our Measurement data is stored in an EAV data model E(Measurement)A(Data_Type)V(Data_Value). We can easily include new measurements by adding new data types to our two lookup table implementation☺:

Table	Columns of table			
Trial	Trial_ID	Purpose		
Measurements	Trial_ID	Measurement_Date		
Measurement_data	Trial_ID	Measurement_Date	Data_Type	Data_Value

Just imagine, if a new trial is started without EAV. And Data1-DataF is not required or have no meaning in the trial. It is required to have DataN…DataZ, but also Data0X…DataT5. I know which approach I will follow and it is not the typical relational theory approach.

How does this all tie up to data-driven programming? Well we said in software development terms that [Abstract]Form = [Abstact]Menu = [Abstract]MenuItem = OBJECT|CLASS. So lets look at common properties.

- All classes have an unique id;
- (Almost) all classes have a descriptive (Text) property
- All classes have 0 or many parameters required at instantiation;
- All classes have properties that describes them in more details;
- All classes have methods or events that do some purpose;
- Not all classes have the same properties, methods and events although some might share the same.

To me this sound like a typical TTLT & EAV model. Back to the drawing board, we need to see if we can implement the TTLT and EAV in our datastore.

5. Changing our datastore to TTLT & EAV

Firstly let us review the datastore used in Article 4 (Listing 7):

Listing 7: Article 4 datastore definition

```
[applicationform]
properties=name:HelloWorldVN;text:Hello World Vulcan Application
controls=menu:mainmenu;datagrid:datagridview

[menu]
mainmenu=text:Main menu of HelloWorldVN application
filemenu=text:File menu of HelloWorldVN application

[menuitem]
mainmenu0=text:&File;eventtype:menu;eventid:filemenu
filemenu0=text:&Hello world;eventtype:eventclick;eventid:menuitemclick
filemenu1=text:How are &you;eventtype:eventclick;eventid:menuitemclick
filemenu2=text:Good&bye world;eventtype:eventclick;eventid:menuitemclick
filemenu3=eventtype:separator
filemenu4=text:E&xit;eventtype:eventclose;eventid:menuitemclose

[datagrid]
```

```
datagridview=A data grid
```

We have an entry point defined as applicationform or in more detail an application containing a form. Form contains member(s):mainmenu and datagridview (in the unknown currently) and some properties:name and text.

It also defines menu, menu item and datagrid. Each containing member(s) and or events that belong to eventtypes, until we reach the bottom or no more members are defined. The end of the chain.

Listing 8: Data store breakdown into smaller components

```
[application]
HelloWorldVN=text:Hello World Vulcan Application;membertypeexec:form;memberid:HelloWorldVN

[form]
HelloWorldVN=text:Hello World Vulcan Application;membertype:menu;memberid:mainmenu

[menu]
mainmenu=text:Main menu of HelloWorldVN application; ~ // ~ means continuation of line
         membertype:menuitem;memberid:menuitemfile
filemenu=text:File menu of HelloWorldVN application; ~
         membertype:menuitem;memberid:menuitemfile; ~
         membertype:menuitem;memberid:menuitemhello; ~
         membertype:menuitem;memberid:menuitemhowyou; ~
         membertype:separator;memberid:separator; ~
         membertype:menuitem;memberid:menuitemexit

[menuitem]
menuitemfile=text:&File;membertype:menu;menuid:filemenu
menuitemhello=text:&Hello world;eventid:menuitemclick
menuitemhowyou=text:How are &you;eventid:menuitemclick
menuitembye=text:Good&bye world;eventid:menuitemclick
menuitemexit=text:E&xit;eventid:menuitemclose

[separator]
separator=text:Menu separator

[event]
menuitemclick=eventtype:eventclick;
menuitemclose=eventtype:eventclose

[eventtype]
eventclick=event:Click;action:add;addtype:EventHandler{owner, @MenuItemClick()}
eventclose=event:Click;action:add;addtype:EventHandler{owner, @MenuClose()}

[datagrid]
datagridview=A data grid
```

When we look at the above we can see it is becoming quite a mess trying to define all the types of classes, members, properties and events. The IniFile concept is also not ideal, but we will have to try and make it work to explain the concept. First things first, lets see if we can identify descriptors that could potentially assist in cleaning up the above making use of the concept of <Identifier>_no, <Identifier>_id, etc.

Classes could be a potential (Listing 9):

Listing 9: Data store class table

```
[class]
1=type:application;id:HelloWorldVN;text:Hello World Vulcan Application;
2=type:form;id:HelloWorldVN;text:Hello World Vulcan Application Form;
3=type:menu;id:mainmenu;text:Main menu of HelloWorldVN application;
4=type:menu;id:filemenu;text:File menu of HelloWorldVN application;
5=type:menuitem;id:menuitemfile;text:&File
6=type:menuitem;id:menuitemhello;text:&&Hello world;
7=type:menuitem;id:menuitemhowyou;text:How are &you
8=type:menuitem;id:menuitembye;text:Good &bye
9=type:menuitem;id:menuitemxit;text:E&xit
10=type:separator;id:separator
```

Our datastore seems to look a lot simpler than before the change, we have defined each class having a unique <Class>_no (internal unique id) and a unique <Class>_id (business unique id). However, looking in more detail, we can see that we have a property "[class]type" that looks like a typical example that we can put in a TTLT (Listing 10).

Listing 10: TTLT lkpdef & lkpitem

```
[lkpdef]
1=lkpdef_id:classtype;lkpdef_description:Types of classes
2=lkpdef_id:eventclick;lkpdef_description:Types of events

[lkpitem]
1=lkpdef_no:1;lkpitem_id:application;default:Y
2=lkpdef_no:1;lkpitem_id:form;
3=lkpdef_no:1;lkpitem_id:menu;
4=lkpdef_no:1;lkpitem_id:menuitem;
5=lkpdef_no:1;lkpitem_id:separator;
6=lkpdef_no:2;lkpitem_id:menuitemclickevent;Default:Y
6=lkpdef_no:2;lkpitem_id:applicationclose;
```

We need to update our class datastore to make use of this reference TTLT (Listing 11).

Listing 11: Data store class table referencing the TTLT (class.classtype_no REF lkpitem.lkpitem.lkpitem_no

```
[class]
1=classtype_no:1;class_id:HelloWorldVN;text:Hello World Vulcan Application;
2=classtype_no:2;class_id:HelloWorldVN;text:Hello World Vulcan Application Form;
3=classtype_no:3;class_id:mainmenu;text:Main menu of HelloWorldVN application;
4=classtype_no:3;class_id:filemenu;text:File menu of HelloWorldVN application;
5=classtype_no:4;class_id:menuitemfile;text:&File
6=classtype_no:4;class_id:menuitemhello;text:&Hello world;
7=classtype_no:4;class_id:menuitemhowyou;text:How are &you
8=classtype_no:4;class_id:menuitembye;text:Good &bye
9=classtype_no:4;class_id:menuitemxit;text:E&xit
10=classtype_no:5;class_id:separator
```

We have defined our classes on an abstract level making use of a TTLT, we still need to associate these classes together via a member list. We call it classmember with classmember.class_no REF class.class_no, membertype_no REF lkpitem.lkpitem_no and class.class_no REF classmember.member_no. The only issue we need to take care of is to not allow recursive class->classmember->class. Additionally we also can potentially have <class|member>type_no referring to a wrong lkpitem_no. We don't worry too much about the details, since we created a database side integrity check to ensure membertype_no can only be of type class_no (Listing 12).

Listing 12: Data store class member table referencing the TTLT

```
[classmember]
1=class_no:1;membertype_no:2;member_no:2;seq=0
2=class_no:2;membertype_no:3;member_no:3;seq=0
3=class_no:3;membertype_no:3;member_no:4;seq=0
4=class_no:4;membertype_no:4;member_no:5;seq=0
5=class_no:4;membertype_no:4;member_no:6;seq=1
6=class_no:4;membertype_no:4;member_no:7;seq=2
7=class_no:4;membertype_no:4;member_no:8;seq=3
8=class_no:4;membertype_no:5;member_no:10;seq=4
9=class_no:4;membertype_no:4;member_no:9;seq=5
```

If we assume the above as a true RDBMS, we can create 4 views to assist us in fetching the details of each (Listing 13):

Listing 13: Class and member views interfacing to TTLT

```
CREATE VIEW lkpitemview AS
 SELECT d.lkpdef_id, lkpdef_description, i.*
  FROM lkpdef d, lkpitem i WHERE d.lkpdef_no = i.lkpdef_no;

CREATE VIEW classview AS
 SELECT c.*, l.lkpitem_id AS classtype_id
  FROM class c, lkpitemview l WHERE c.classtype_no = l.lkpitem_no;

CREATE VIEW memberview AS
 SELECT m.*,
        p.classtype_no, p.classtype_id, p.class_id AS member_id,
        l.lkpitem_id AS membertype_id
  FROM classmember m, p.classview p, lkpitemview l
   WHERE m.member_no = p.class_no AND m.membertype_no = l.lkpitem_no;

CREATE memberownerview AS
 SELECT m.*,
        o.class_no AS owner_no, o.class_id AS owner_id,
        o.classtype_no AS ownertype_no, o.classtype_id AS ownertype_id
  FROM classmemberview m, classview o
   WHERE m.class_no = o.class_no;
```

Not sure how many have seen some possibilities with the above views, but we can actually ask some questions:

- Show me all classes;
- Show me all applications;
- Show me all forms;
- Show me all menus;
- Show me the (tree of) members of class;
- Show me the (tree of) owners of class.

In future articles a bit more discussion on that. We still have not yet address EAV and where we can implement it. If we look at the class store, Properties have all some features that is shared between all of them. However, although separator when we use our WED shows it contains a property Text = "-", have tried numerous times to change it to something different, e.g. "+", "|", it appears to have no effect, even if left blank. There is our first clue, it seems

that Text is not shared between all classes. Lets see how we can enhance our datastore via EAV to potentially eliminate this sparse data.

Well effectively we have already implemented an EAV model. Our OTLT that we enhanced to make use of two tables are in effect already conforming to the EAV model. We do not have a single table:class which have columns for App, Menu1, Menu2, Event1, Event2, etc. We know an [App]class might have AppForm. But also we don't know how many Menus an [App]Form might have, or if it would actually have other MemberTypes. A single table will mostly contain sparse data. By adding some additional overhead we will only store items that are of meaning, no null value columns. Our table:classmember is also in effect and EAV implementation. It will only contain records applicable to the table:class. However, we defined above that not all Properties have meaning, e.g. Separator does not need a Text value, but the other classes do. We can jump the gun, and say: Well it's easy, we can extend our classmember table to have a record, not only for members, but also for properties. Mission accomplished! Yes that would be a solution, however my believe is that I would like as far as possible to get all details (properties) of a class in a single row. Otherwise we end up adding complexity since we need to pivot records into a single row. Let's look at our datastore format (Listing 14), since it provides us with a hint.

Listing 14: Data store format

```
[class]
1=classtype_no:1;class_id:HelloWorldVN;text:Hello World Vulcan Application;
```

It contains a typical ini structure

<Key>=<Value>

all stored on a single line (row). I have however sub-itemised it with

<SubKey>:<SubValue>;

We therefore have a structure

<[Sub]Key>=<Value|<<SubKey>:<SubValue>; …>>>

It does not matter what [un]used character we will use, as long as we are consistent and our data store interface knows about it and how to interpret it. I will make use of "#" for <Key>#<Value> separator and "!" for end of subkey value pairs. Lets implement our subkey for sparsely filled properties called classproperty (Listing 15):

Listing 15: Data store with sparsely filled properties format

```
[class]
1=classtype_no:1;class_id:HelloWorldVN; // Continue on next line
                 classproperty:text#Hello World Vulcan Application!
                               nextprop#somevalue;
                 class_description:Hello world application in Vulcan
```

You probably ask why we doing this, surely we could just have text and nextprop on the same level without the complexity of another sub-level?

Yes unfortunately I hav to agree that is the case, in a real life situation we would however not have the flexibility of a ini interface.

Lets look at this from a database table perspective, it might make it clearer. The class table would probably look something like:

Table 8: Database class table structure

Table:Class					
Class_no	Classtype_no	Class_id	Text	Nextprop	Class_description
1	1	HelloWorldVN	Hello World Vulcan Application	Some value	Hello World Application in data-driven application build with Vulcan.NET

As we have stated, Text and Nextprop will potentially be sparsely filled, or for the moment be in the unknown of class properties. And here is where a variation on the EAV model (1 record describe 1 class) comes into play (Table 9).

Table 9: Database class table structure modified for EAV

Table:Class				
Class_no	Classtype_no	Class_id	Class_description	Class_property
1	1	HelloWorldVN	Hello World Application in data-driven application build with Vulcan.NET	Text=Hello World Vulcan Application; nextprop= Some value

Now the only work we will have to do is to somehow tell our interface layer that it contains an EAV column named Class_property and treat each item inside the same as if they a column of the table!!! ☺

6. Summary

I hope this article gave some readers maybe an idea of how to implement those “difficult” database design issues. We have effectively try and eliminate some of the issues of One True Lookup Tables and showed how we can enhance the EAV model.

In the next article we will be adapting our application datastore interface accordingly. Sorry no little application source code in this series. In the next article we will make up for that!

Till the next article: Creating a datastore agnostic interface layer