Data-driven Programming

By

Johan Nel

A series of articles explaining the principles

Article 4: The application menu

November 2014

Table of contents

Listings

1. Back to our application menu

Let's take the lessons learned in the design of our CustomerView class in article 3 and see if we can apply it to our application and specifically looking at the menu. In this article we will be looking at extracting the details (data) that reside in our application menu and how we can populate it at runtime. Unfortunately we need to get a bit off topic, since we will address other issues too. In the first article it was stated that programming should be done at an abstract level. We will therefore start with initial code and hopfully show how we can move to this abract code level.

Lets get started!

2. Data is best stored in a well designed relational database

First question we ask is what data describe a menu? We look at the code generated for our menu and start extracting the data components. In this context we state that our application can have 0 or 1 main menu. MainMenu can have 0 or many MenuItems. Each of MenuItem can have other MenuItems belonging to it or otherwise stated, a MenuItem can be the owner to other Menus, or MenuItem can perform a certain task, let's call it MenuEvent. MenuItems can also be sub-grouped by a special menu item type called a MenuSeparator with the only purpose of spacing our related MenuItems, instead of grouping them in a sub-menu.

Our first task would be to design an external store for our menu data, based on the above statements using basic relational theory (Table 1 and Table 2). Each menu will have a unique name and some description that allow any reader to understand the purpose in database language. Each menu item needs to belong to only 1 menu (we will later contradict this thohg), it need to have an order in the list on the same level, a display value and it can be one and only one of: the owner of another menu, be a divider or perform a process.

Table 1: Menu definition table

ColumnName	**Column Type**	**Column Width**	**Column Dec**	**Primary key (PK)** **Foreign key (FK)**
MENU_ID	C	50	0	PK
DESCRIPT	C	100	0	

Table 2: Menu item table

ColumnName	Column Type	Column Width	Column Dec	Primary key (PK) Foreign key (FK) Unique key (UK)
MENU_ID	C	50	0	PK, FK1
SEQ	N	3	0	PK
DISPLAY	C	50	0	
EVENTTYPE	C	50	0	
EVENTID	C	50	0	FK2

In the next step we will try and populate our datatables with rows based on what we see in our IDE designed MenuStrip (Table 3 and Table 4) to describe our menu in relatational data language.

Table 3: Menu definition data

MENUID	DESCRIPT
mainmenu	Main menu of HelloWorldVN application
filemenu	File menu of HelloWorldVN application

Table 4: Menu item data

MENUID	SEQ	DISPLAY	EVENTTYPE	EVENTID
mainmenu	0	&File	menu	filemenu
filemenu	0	&Hello world	eventclick	menuitemclick
filemenu	1	How are &you	eventclick	menuitemclick
filemenu	2	Good&bye world	eventclick	menuitemclick
filemenu	3	-	separator	
filemenu	4	E&xit	close	closeclick

As example we will use the String File Data Management System. The format was put in the public domain and MicroSoft decided to create there own version of StringPad, known as NotePad that was published as open source. With our new tools we start creating our first Menu Data Management Store and call it HelloWorldVN.exe.menu.

The results of our efforts are shown in Listing 1.

Listing 1: Populated "database" with menu and menu item data

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

[menuitem]
mainmenu0=display:&File;eventtype:menu;eventid:filemenu
filemenu0=display:&Hello world;eventtype:eventclick;eventid:menuitemclick
filemenu1=display:How are &you;eventtype:eventclick;eventid:menuitemclick
filemenu2=display: Good&bye world;;eventtype:eventclick;eventid:menuitemclick
filemenu3=display:-;eventtype:seperator;eventid:
filemenu4=display:E&xit;eventtype:eventclick;eventid:closeclick
```

We now have a repository where we can manage and ask questions about our HelloWorldVN menus. No need to look at source anymore. Anybody with NotePad can find the info they need regarding HelloWorldVN menus (using <Ctrl>F). At the moment it is not of much use though, we created a new task that needs to be maintained on top of our already overloaded job, we need to maintain the menu inside of our application and we need to maintain our menu data repository.

3. Making our menu class data augnostic

So let us look at making some changes. Firstly we need to write some code, sorry no drag and drop designer here. We software developers, not relying on somebody else doing the dirty work for us. First step will be to create a class and we can use our HelloWorldVN BasicForm as a start and change or remove code that is not applicable to our MenuStrip. We end up with a menu shell as in Listing 2.

Listing 2: Empty ddMenu class

```
CLASS ddMenu INHERIT System.Windows.Forms.MenuStrip

  CONSTRUCTOR(sName AS STRING)
     SUPER()
     IF sName:Length <= 0
        sName := "ddMenu"
     ENDIF
     SELF:InitializeMenu(sName)
  RETURN

  METHOD InitializeMenu(sName AS STRING) AS VOID
     SELF:Name := sName
     //Load menu items
```

```
  RETURN
END CLASS
```

We have successfully implemented the basic framework for our menu system. It contains a Name data property and we effectively did not hard code it, although not external to the application, we can state that our menu will receive the data at runtime and not compile time. In the event of our menu not receiving any data, we auto-populate it. For all practical purposes we can tell our application form to consume our class via an object. It will be displayed on the form. Mission accomplished!

4. The application form

We can therefore modify our application form to tell it to make use of this new agnostic menu feature. So let us write some code again, or use our WED tool to do some work and strip out the details that we don't need (Listing 3).

Listing 3: Skeleton form

```
CLASS ddAppForm INHERIT System.Windows.Forms.Form

CONSTRUCTOR()
  SUPER()
  SELF:InitializeForm()
RETURN

  METHOD InitializeForm() AS VOID

     SELF:SuspendLayout()

     SELF:ClientSize := System.Drawing.Size{392 , 264}
     SELF:Name := "ddAppForm"
     SELF:Text := "Data-driven application form"

     SELF:Controls:Add(ddMenu{“mainmenu”})

     SELF:ResumeLayout()

  RETURN
END CLASS
```

Something is however not right. We still have hard coded data in our ddAppForm. And our form has one function: To receive a menu and do something with it. Is it our form doing it or is the menu clever enough to do it? We don't know but we don't care at the moment actually. Our form's task is to contain a menu and let it do its purpose. Lets see if we can rectify the situation. We determined that our form need a menu, we need to add something that will tell the form that it needs a new member of type menu. We can define a way for the form to fetch menus. However with some foresight we also identify that the form might need other types of members, not only menus. Does the form know what members it will have? No, it only

knows that it might have members of which menu we known of, there might also be some in the unknown. If we look at our initial WED designed code it is doing some replication. When we hear the word replication we need to think LOOP. So with this in mind let us sub group this task. We call it MemberBuild instead of MenuGet/Fetch/Build and enhance our form accordingly (Listing 4). We also determine how MemberBuild will work by inspection of the WED form that we create initially for HelloWoldVN. To make it consistent, we observe that our initial form has a collection property that members were added to, called Controls. If we research MenuStrip we see that it appears all members will somehow have a base class they inherit from Control. That gives the first hint to what we need to do and we can generate some generic code. Our terminology is however inconsistent, MemberBuild is less descriptive of what we doing, so we replace it with ControlsAdd (Listing 4).

Listing 4: Adding controls to HelloWoldVN application form

```
METHOD InitializeForm() AS VOID
  …
  SELF:ControlsAdd()
  …
RETURN

METHOD ControlsAdd() AS VOID
  LOCAL lstControls AS List<Control>
  lstControls := List<Control>{}
  lstControls:Add(ddMenu{"mainmenu"})
  FOREACH ctrl AS Control IN lstControl
     SELF:Controls:Add(ctrl)
  NEXT
RETURN
```

Houston we have trouble, our form can add controls, but it need some method of identifying them, there might be more than a mainmenu. So our form need to do some shouting: "Hello Controls of the VN World, I have a container and I need to fill it up!!!". The biggest problem we still have is that although the form has empty shelvespace, it needs to request products from the supplier to fill them. A supermarket is not a supermarket if it is a building with empty shelves. We need to advertise to our suppliers that we need stock, the how is however unknown. The data "mainmenu" is stil hard coded and we anticipate we will not only stock "mainmenu". We can remove it and move it higher up the tree, however we just passing the buck. We want to do it once to make it persistent to our data-driven rule: Data should reside outside of the application, it is not hard coded.

So we need to gather some data of what we want to sell, if we a bakery, we sell bread, not meat. That we will leave to the butcher. Bread needs some ingredients, however what those ingredients are, we don't care it is the supplier's responsibility. Same with our form class it needs some data to define what we want in our collection of controls. We defined that data

resided outside of an application. It is not hard coded. So lets see if our String Data Management System Repository has the details (Listing 1). There is no descriptive information that we can see. We quickly realise our data store have some flaws, it constains a missing link, we need to enhance it. So off we go and brainstorm with our data administrator what we need. The following list all the requirements identified:

- Our form need some data to describe properties;
- Our application form needs a way to identify what controls it need to collect;
- We need to be able to associate the form with it's collection of controls;

Our DBA gave the requirements some thought and came up with the following solution:

- We should enhance the datastore not to only include menus. A new name is required, since it contain some features other than menu: HelloWorldVN.exe.db;
- Add an additional substorage to describe our application form;
- Create a data property telling the form it has a control type menu and that the menu is identified by mainmenu;
- Some properties of application form are hardcoded and we need to define them in our external data store
- We need a method to communicate between our data store and form;

The new datastore contain the following after the enhancements were made (Listing 5).

Listing 5: Enhanced data store including application detail

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

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

[menuitem]
mainmenu0=display:&File;eventtype:menu;eventid:filemenu
filemenu0=display:&Hello world;eventtype:eventclick;eventid:menuitemclick
filemenu1=display:How are &you;eventtype:eventclick;eventid:menuitemclick
filemenu2=display: Good&bye world;;eventtype:eventclick;eventid:menuitemclick
filemenu3=display:-;eventtype:seperator;eventid:
filemenu4=display:E&xit;eventtype:eventclick;eventid:closeclick
```

5. The data interface layer

We now need a way for our form to communicate with the data store. After doing some extensive research, we discover that on FacelessTrading a product is available that can communicate between our data store and our application called jhnIniFile. Knowing a private investigator, we call Magnum PI and request to use his connections to find out if this product

can deliver the goods. He immediately makes contact with the Internet Investigation Agency (IIA) and they determine that it appears to be a good product. It shows a value of 100 browser hits (BH) and 85 download hits (DH) with no return (of downloads) hits (RH). We decide to spend some of our hardearned Personal Hits (PH) on the product and in no time we are comfortable that it was a good investment.

It is time to get back to our development environment and become productive.

6. Form = Menu

As efficient software developers we review our code on a regular basis. However we all look at code in different ways. I will share my view on our AppForm and Menu hopefully showing what I meant by abstract programming.

I can summarize it with the pseudo-code in ().

Listing 6: Pseudo class layout

```
CLASS <ClassName>[ INHERIT <BaseClass>]
  CONSTRUCTOR(<ClassProperties>)
    SUPER()
    SELF:SetDefaultsForClassProperties
    SELF:Initialize<ClassName>(<ClassProperties>)
  RETURN

  METHOD SELF:Initialize<ClassName>(<ClassProperties>) AS VOID
     FOREACH property IN <ClassProperties>
        SELF:<Property> := property
     NEXT
     FOREACH member IN <MemberCollection> // We can put this in MembersAdd method if required
        SELF:<MemberCollection>:Add(member)
     NEXT
  RETURN

END CLASS
```

7. Summary

I hope this article in the series gave enough insight into how we look at code from a data-driven perspective. In the next article we will look at our interface between the client (application and menu) and datastore. We will look at an interface layer and also at ways we can write code once and re-use many times.

Till the next article: The client<->data interface.