Data-driven Programming

By

Johan Nel

A series of articles explaining the principles

Article 5: Abstracting the data interface layer

November 2014

Table of contents

1. WHAT WE HAVE LEARNED 1

2. GETTING THE DATA INTO OUR APPLICATION 1

3. IDENTIFY REPLICATED CODE, ABSTRACT AND RE-USE 3

4. DATA INTERFACE LAYER 4

5. ABSTRACTING OUR INTERFACE LAYER 6

6. SUMMARY 8

Listings

LISTING 1: PSEUDO CLASS LAYOUT 1

LISTING 2: PSEUDO MENU CLASS LAYOUT 2

LISTING 3: EXAMPLE STRING REVERSAL PROCESS 3

LISTING 4: STRING REVERSAL FUNCTION 4

LISTING 5: INTERFACE BETWEEN PRESENTATION LAYER AND MEMBERS 7

Figures

FIGURE 1: HELLO WORLD APPLICATION AND MEMBER DETAIL 3

FIGURE 2: HELLOWORLDVN WITH A MENU STRIP 7

FIGURE 3: COMPLETED HELLOWORLDVN APPLICATION 8

1. What we have learned

We have set the foundation for our methodology in the previous articles. We also learned that presentations (User Interfaces) have similarities between them and we therefore can make a statement that AbstractForm = AbstractMenu. In the previous article we defined a method for storing our presentation layer data in a datastore. We determined that we could use a tool (jhnIniFile) to get that data into our application. We now need something to present this data to our classes.

Lets get developing!

2. Getting the data into our application

In article 4 we identified that it is of no use to present our menu data in isolation. We enhanced our datastore to also contain the data of application and we told application data that it contains a member with name menumain that is of type menu. To access our datastore we need to tell jhnIniFile where it resides. A logical way to do this is to make use of the internal Application.ExecutablePath property. Our jhnIniFile requires the extention of the datastore to be “ini”. To make our application(s) datastore aware, we use the convention of “<Location><Application>.exe.ini”. It provides us with a consistent interface, as long as we ensure our application and datastore resides in the same location (Listing 1).

Listing 1: Pseudo class layout

```
CLASS ddForm INHERIT System.Windows.Forms.Form
  CONSTRUCTOR()
     SUPER()
     SELF:InitializeForm()
  RETURN

  METHOD InitializeForm() AS VOID
     LOCAL oIni AS jhnIniFile
     oIni := jhnIniFile{Application.ExecutablePath + ".ini"}
     SELF:Name := oIni:GetString("applicationform", "name")
     SELF:Text := oIni:GetString("applicationform", "text")
     SELF:SuspendLayout()
     SELF:ControlsAdd()
     SELF:ResumeLayout()
  RETURN

  METHOD ControlsAdd() AS VOID
     LOCAL oIni AS jhnIniFile
     LOCAL sControls AS STRING
     LOCAL aControls, aKV AS STRING[]
     LOCAL lstControls AS SortedList<STRING, STRING>
     oIni := jhnIniFile{Application.ExecutablePath + ".ini"}
     sControls := oIni:GetString("applicationform", "controls")
     aControls := sControls:Split(";":ToCharArray(), StringSplitOptions.RemoveEmptyEntries)
     lstControls := SortedList<STRING, STRING>{}
     FOREACH ctrldef AS STRING IN aControls
        aKV := ctrldef:Split(":":ToCharArray())
        IF aKV:Length > 0 .AND. aKV[1]:Length > 0
           lstControls:Add(aKV[0], aKV[1])
```

```
      ENDIF
   NEXT
   FOREACH key AS STRING IN lstControls:Keys
      aControls := oIni:GetSection(key)
      FOREACH ctrl AS STRING IN aControls
         IF ctrl:StartsWith(lstControls:Item[key])
            MessageBox.Show(ctrl)
            // Add member to Controls via some interface
            // SELF:Controls:Add(SomeInterface(ctrl))
         ENDIF
      NEXT
   NEXT
 RETURN

END CLASS
```

In the previous article we have said AbstractApp = AbstractMenu. Lets give it a try and see if we can use copy and replace methodology to change our form class into a menu class (Listing 2).

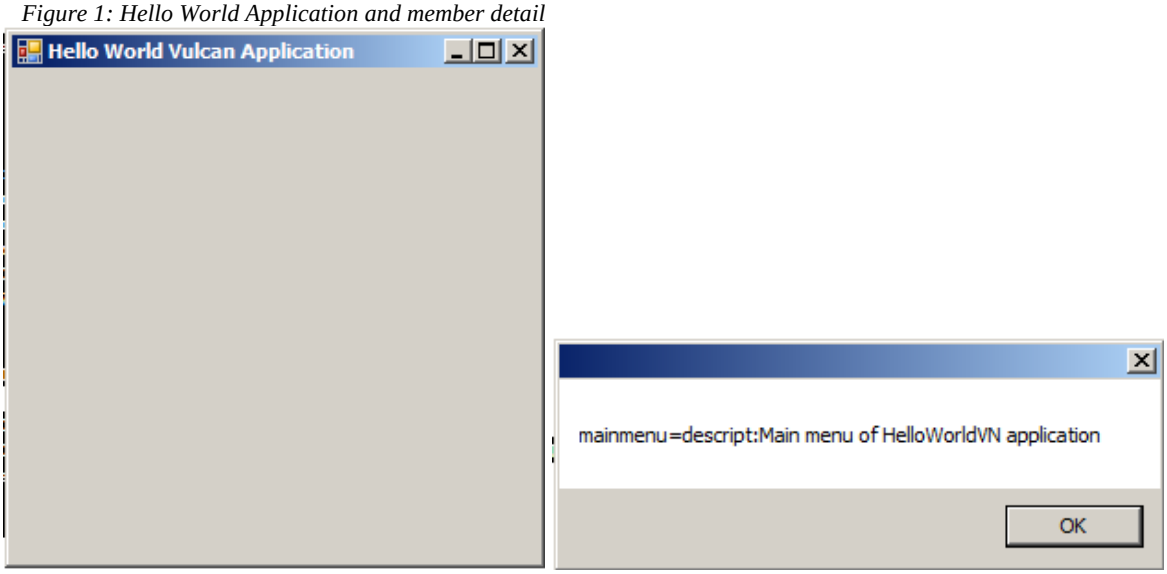
Listing 2: Pseudo menu class layout

```
CLASS ddMenu INHERIT System.Windows.Forms.MenuStrip
     CONSTRUCTOR()

        SUPER()
        SELF:InitializeMenu()

     RETURN

     METHOD InitializeMenu() AS VOID
        LOCAL oIni AS jhnIniFile
        oIni := jhnIniFile{Application.ExecutablePath + ".ini"}
        SELF:Name := oIni:GetString("applicationform", "name")
        SELF:Text := oIni:GetString("applicationform", "text")
        SELF:SuspendLayout()
        SELF:MenuItemsAdd()
        SELF:ResumeLayout()
     RETURN

     METHOD MenuItemsAdd() AS VOID
        LOCAL oIni AS jhnIniFile
        LOCAL sControls AS STRING
        LOCAL aControls, aKV AS STRING[]
        LOCAL lstControls AS List<STRING>
        oIni := jhnIniFile{Application.ExecutablePath + ".ini"}
        sControls := oIni:GetString("applicationform", "controls")
        aControls := sControls:Split(";":ToCharArray(), ;
                                     StringSplitOptions.RemoveEmptyEntries)
        lstControls := List<STRING>{}
        FOREACH ctrldef AS STRING IN aControls
           aKV := ctrldef:Split(":":ToCharArray())
           IF aKV:Length > 0 .AND. aKV[1]:Length > 0 .AND. aKV[0] == "menu"
              lstControls:Add(aKV[1])
           ENDIF
        NEXT
        FOREACH key AS STRING IN lstControls
           LOCAL sProperty := oIni:GetString("menu", key) AS STRING
           MessageBox.Show(sProperty)
           // Add member to Controls via some interface
           // SELF:Controls:Add(SomeInterface(sProperty))
        NEXTRETURN
END CLASS
```

Looking at the above code, although there are some anomalies that will not allow this to to be called, I believe we have done 90%+ of the job. The menu class still contain some information that should be part of the application form class and our application form know it need to add something via SomeInterface, but it is not yet implemented. We have only

achieved to be able to tell our AppForm that is has a name of “HelloWorldVN” and a text value of “Hello World Vulcan Application”. It is also able to define it will have a member “mainmenu” and that is has some properties which is of no relevance to AppForm, but it need to pass that on to something that will know what to do with it (Figure 1).

Figure 1: Hello World Application and member detail



Looking again at our code we can identify that although we created an interface between our datastore, AppForm and Menu via jhnIniFile, it seems we replicating again. What if for some reason we change the format of our datastore? We will need to edit all places that our interface is implemented. If we lucky, we will adapt where applicable to make the changes. It might also happen that we overlook a change required and injecting our application with a bug or a behaviour that we don’t expect.

3. Identify replicated code, abstract and re-use

When we need to convert a string into reverse order one way of doing it would be (Listing 3):

Listing 3: Example string reversal process

```
sString := “Hello world”
nLen := sString:Length
oStack := Stack<STRING>{}
FOR LOCAL i := 0 UPTO nLen – 1
  oStack:Push(sString[i])
NEXT
sString := “”
WHILE oStack:HasEntries()
  sString += oStack:Pop()
ENDDO
```

This is fine, however if we need to do it many times in our application, we abstract it (Listing 4):

Listing 4: String reversal function

```
FUNCTION ReverseString(s AS STRING) AS STRING
  LOCAL s := "" AS STRING
  LOCAL oStack AS Stack<STRING>
  nLen := s:Length
  oStack := Stack<STRING>{}
  FOR LOCAL i := 0 AS INT UPTO nLen – 1
     oStack:Push(sString[i])
  NEXT
  WHILE oStack:HasEntries()
     s += oStack:Pop()
  ENDDO
RETURN s

sString := "Hello world"
sString := ReverseString(sString)
```

What we currently have is a persistent interface between our data store and our application via jhnIniFile. However we replicating how we interface between jhnIniFile and our presentation layer, AppForm and [App]Menu. It is doing the same in both.

4. Data interface layer

We do not want to hardcode the interface to our presentation layer. It means everytime there is a change in the datastore format we need to go through each presentation layer and ensure it is adapted to behave correctly.

In abstract terms we would define the path between datastore and presentation layer currently to be:

DataStore->PresentationLayer(including Internal interface)

We would rather have:

DataStore->Internal interface->PresentationLayer

It is also possible that we will have to provide various formats for the Internal interface to pass our data through to the PresentationLayer. Can we abstract (generalise) it further? Yes we can:

DataStore->Internal interface->Universal presentation->PresentationLayer

We have a defined DataStore:HelloWorldVN.exe.ini, we have an interface into our application:jhnIniFile, and we have PresentationLayer:AppForm and AppMenu, but they both contain the Interface to our datastore. We need to effectively communicate between InternalInterface and PresentationLayer, a missing or is it a replicated link...

Over the years I have made many mistakes regarding my implementation. Yes they all worked, however as one keep on enhancing applications, sometimes you wonder looking at

code, how could you have done it that way. What I would try and do is not to go into my mistakes, but rather jumpstart anybody that want to implement data-driven concepts to what I believe is currently the bullet proof method, it does not mean missile proof though. It would however help if people have ideas where they believe I am wrong and contribute by sharing them.

Lets look at our Internal interface. We have jhnIniFile everywhere interfacing our datastore to our presentation layer. We want to remove that. I will jump in at the deep end and see how successful my swimming lessons were.

From what we can see our datasource contain descriptive data regarding a class, but it also contain a container that objects of other or the same class can be added to, members of the parent class/object. If we look at the 3 types of classes our application have AppForm, Menu and MenuItem we again try to describe them on an abstact level (Table 1).

Table 1: Potential members of our 3 classes in HelloWorldVN

Class	**Unique Identifier**	**Properties**	**Potential Members**	**Events**
[ddApp]Form	Name	Text, Size, Origin, …	[dd]Menu[s], Unknown	Unknown
[dd]Menu	Name	Text,…	[dd]MenuItem[s]	Unknown
[dd]MenuItem	Name	Text, …	MesageBox.Show(), Unknown	Associate event with potential member[s]

In abstract terms we therefore state that most of our Containers (main classes) can have members, a typical Parent-Child relation. Application has Menus and maybe other members. Menus have MenuItems, MenuItems can MenuItems or do a process. Somewhere the Parent->Child will stop but we have many and it is almost impossible to maintain them. We will look at this in the next article if we can modify our DataStore to guarantee future expansion without us having to modify the structure and repercussions in our application.

Back to our source code.

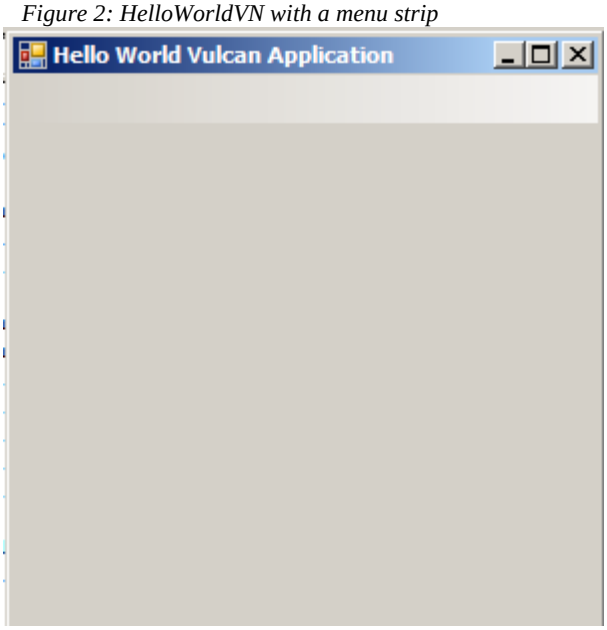
5. Abstracting our interface layer

The first problem if we remove the direct interface to jhnIniFile in our classes is how to effectively tell it what we need. Second problem we encounter, how will the Interface know where to add the members to? Well let's start with the adding of members first. Delegates are the answer, so lets see if we can do something about it. It appears that we will add Controls to a form, and ToolStrip[Menu|Separator]Item to [App]Menu. Unfotunately both types don't inherit from Controls, so we need to do some (Cast)o in our code. Following our naming convention we have ControlsAdd(), which imply we going to add Controls. Similarly we adding ToolStripItems or let us call it MenuItems and a MenuItemsAdd() which implies we going to add a list of items to both. So how are we going to add individual items? As said, delegates to the rescue. We can tell whoever need/want to know where to add them, lucky we have identified who our Big Brother who knows all is: ddMemberInterface:

```
DELEGATE MemAdd(o AS OBJECT) AS VOID

////////// AppForm //////////
METHOD ControlsAdd() AS VOID
  LOCAL delCtrlAdd AS MemAdd
  LOCAL oMbrInt AS ddMemberInterface // Who are we going to request our members from
  delCtrlAdd := MemAdd{SELF, @ControlAdd()}
  oMbrInt := ddMemberInterface{}
  FOREACH ctrl in List of Controls of AppForm
     oMbrInt:GetMeMyControls(<what to look for>, delCtrlAdd)
  NEXT
RETURN

METHOD ControlAdd(o AS OBJECT) AS VOID
  SELF:Controls:Add((Controls)o)
RETURN

////////// ddMenu //////////
METHOD MenuItemsAdd() AS VOID
  LOCAL delMemAdd AS MemAdd
  LOCAL oMbrInt AS ddMemberInterface
  delMemAdd := MemAdd{SELF, @MenuItemAdd()}
  oMbrInt := ddMemberInterface{}
  FOREACH item IN List of ToolStripItems of ddMenu
     oMbrInt:GetMeMyMenuItems(<what to look for>, delMemAdd)
  NEXT
RETURN

METHOD MenuItemAdd(o AS OBJECT) AS VOID
  SELF:Items:Add((ToolStripItem)o)
RETURN
```

With the interface in place for out AppForm our HelloWorldVN application miraculously suddenly contain a member: MenuStrip (Figure 2).

Figure 2: HelloWorldVN with a menu strip



Well in our next step we will have to create the Big Brother. We move our Internal Data Store (jhnIniFile) to Big Brother and we provide a method to add our members to our classes. To know where the member is added to, we pass in our delegate. Hey Big Brother, I need you to get me a member! I don't know how he looks like, but I know his name, here is the details of the member, but by the way you don't know who I am, just listen to what I need and give it to me via my mediator MemAdd (Listing 5).

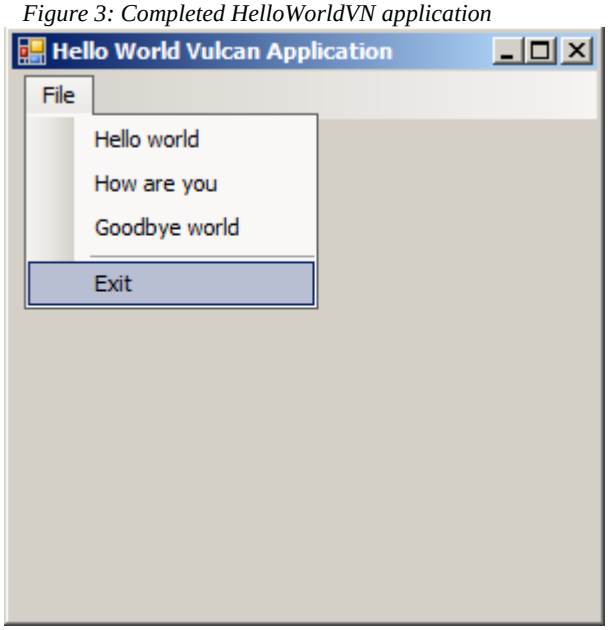
Listing 5: Interface between presentation layer and members

```
SEALED CLASS ddMemberInterface
  STATIC HIDDEN _inst AS ddMemberInterface
  HIDDEN oIni AS jhnIniFile

  STATIC CONSTRUCTOR()
     _inst := ddMemberInterface{}
  RETURN

  HIDDEN CONSTRUCTOR()
     SUPER()
     SELF:oIni := jhnIniFile{Application.ExecutablePath + ".ini"}
  RETURN

  STATIC PROPERTY Inst AS ddMemberInterface
     GET
        RETURN _inst
     END GET
  END PROPERTY

  METHOD MemberAdd(mbrdet AS STRING, memadd AS MemAdd) AS OBJECT
     LOCAL o AS OBJECT

     IF mbrdet:StartsWith("menu:")
        o := ddMenu{mbrdet}
        memadd(o)
     ELSEIF mbrdet:StartsWith("menuitem:")
        IF mbrdet:Contains("eventtype:separator")
           o := jhnToolStripSeparator{mbrdet:Replace("menuitem:", "")}
        ELSE
           o := ddMenuItem{mbrdet:Replace("menuitem:", "")}
        ENDIF
        memadd(o)
     ELSE
        MessageBox.Show("Unknown object type : " + mbrdet, "ddMemberInterface")
     ENDIF
```

```
   RETURN o

   METHOD PropertyGet(owner AS STRING, prop AS STRING) AS STRING
      LOCAL sProp AS STRING
      sProp := SELF:oIni:GetString(owner, prop)
   RETURN sProp

   METHOD PropertiesGet(owner AS STRING) AS STRING[]
   RETURN SELF:oIni:GetSection(owner)

END CLASS
```

Our application again appeared to have changed ().

Figure 3: Completed HelloWorldVN application



6. Summary

I hope this article in the series gave enough insight into how we look at code from a data-driven perspective. We have basically recreated the Hello World Application in data-driven terms. In the next article we will look at our interface between the client (application) and datastore (<Application>.exe.ini) and if there is not a way we can make it persistent, without having to change the structure if we need new types. For this we will look at the one lookup table (OLT), Entity Attribute Value EAV model and a way to use some of the benefits of that in a relational environment. Happy playing with your data-driven Hello World Application☺

Included in this article you will find all the source code for HelloWorldVN. Please create a ddApp application and maybe step through it with the debugger to see how it works. Happy debugging☺

Till the next article: EAV and OLT principles