Data-driven Programming

By

Johan Nel

A series of articles explaining the principles

Article 7: A persistent interface layer

December 2014

Table of contents

1. INTRODUCTION 1

2. LET THE CODING START 1

3. CONFIGURATION DRIVER....1

4. APPLICATION DRIVER 2

4.1 THE INTERNAL PUBLIC METHOD EXEC() 3
4.2 THE HIDDEN METHOD EXEC(IAPP) 4

5. THE CLASS MEMBER INTERFACE LAYER 5

5.1 JHNMEMBERINTERFACE:MEMBERADD(<PARAMETER COLLECTION>) 7
5.2 JHNMEMBERINTERFACE:MEMBERADD(<PARAMS>, <DELEGATE>)....7

6. THE DATA-DRIVEN APPLICATION START 8

7. THE DATA-DRIVEN APPLICATION FORM 9

8. THE DATA-DRIVEN MENU 10

9. THE DATA-DRIVEN PARAMETER COLLECTION CLASS....12

10. THE DATA-DRIVEN SETUP DICTIONARY CLASS 14

11. SUMMARY 14

Listings

LISTING 1: MODIFIED START() FUNCTION 1
LISTING 2: CONFIGURATION DRIVER CLASS 2
LISTING 3: APPLICATION DRIVER CLASS 3
LISTING 4: JHNAPPLICATIONDRIVER:EXEC() 4
LISTING 5: START SECTION ADDED TO DDFRAMEWORK.EXE.INI 4
LISTING 6: JHNAPPLICATIONDRIVER:EXEC(IAPP) 5
LISTING 7:THE CLASS MEMBER INTERFACE: JHNMEMBERINTERFACE 6
LISTING 8: JHNMEMBERINTERFACE:MEMBERADD(OPC) 7
LISTING 9: JHNMEMBERINTERFACE:MEMBERADD(OPC, MEMADD) 8
LISTING 10: THE DATA-DRIVEN APPLICATION CLASS 9
LISTING 11: THE DATA-DRIVEN APPLICATION FORM 10
LISTING 12: THE DATA-DRIVEN MENU CLASS 11
LISTING 13: THE DATA-DRIVEN MENU ITEM CLASS 11
LISTING 14: THE DATA-DRIVEN TOOLSTRIP SEPARATOR CLASS 12
LISTING 15: THE DATA-DRIVEN PARAMETER COLLECTION CLASS 13

1. Introduction

In article 5 we have created our first working data-driven application. In article 6 we looked at making full use of One (True) Lookup Table, but extending it a bit to suite our needs. We also looked at Entity Attribute Value and how we could keep the concept of 1 record per feature with a text column holding Key=Value[;Key=Value[,…]] lists of sparse data. In this article I will hopefully excite you with another working example (with source code) based on the principle. It will also be the last article that we will be using our acquired jhnIniFile exclusively. We have discovered a new technology (RDBMS) with an enhanced interface compared to NotePad. To the point, we need to do way too much coding to achieve a solution. We will try to aim for classes that can do anything in (almost always) less than 50 lines of code☺ Yes! you heard, sorry read correctly, 50 lines of code per class.

2. Let the coding start

You probably guessed right. We need a Start function, difference is that we need some driver and cannot do the typical Application.Run(oForm1), our framework don't know it. Let's get started and modify our Start function (Listing 1):

Listing 1: Modified Start() function

```
[STAThreadAttribute];
FUNCTION Start( asCmdLine AS STRING[] ) AS INT
  LOCAL nExitCode AS INT
  System.Windows.Forms.Application.EnableVisualStyles()
  System.Windows.Forms.Application.DoEvents()
  jhnFT.Utils.Config.jhnConfigurationDriver.Inst:Exec()
RETURN nExitCode
```

We have a new feature and it seems very similar to how we will do a Console application. Off we go to configure our Application Framework.

3. Configuration driver

There are a couple of things we want our Framework to do:

1. We don't want surprises happening, e.g. having more than 1 instance of our configuration. Singleton to the rescue;
2. We don't want any interference to our configuration, except if we allow it. INTERNAL and SEALED;

3. Our configuration should only be executed once per application (Framework) session, since it might change the behaviour of active events and they might not behave nicely afterwards. STATIC LOCAL to the rescue.

Our configuration driver looks pretty cool. We achieved less than 50 lines of code (to be used somewhere else☺). Basically the configuration driver is an entry into the jhnFT.Utils.Config namespace. Only 1 instance can be active per application and it has only an Exec() method that is public (Listing 2) that execute a Singleton class jhnApplicationDriver:Exec() method .

Listing 2: Configuration driver class

```
BEGIN NAMESPACE jhnFT.Utils.Config
  SEALED CLASS jhnConfigurationDriver
      STATIC HIDDEN _inst AS jhnConfigurationDriver
      STATIC CONSTRUCTOR()
         _inst := jhnConfigurationDriver{}
      RETURN

      HIDDEN CONSTRUCTOR()
         SUPER()
      RETURN

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

      METHOD Exec() AS VOID
         jhnApplicationDriver.Inst:Exec()
      RETURN

  END CLASS
END NAMESPACE
```

We are getting closer to presenting our application. On to the ApplicationDriver class.

4. Application driver

Well I probably do not have to tell what this is all about. We need to somehow get the framework to know how to display our application. In the configuration driver classs we called jhnApplicationDriver.Inst:Exec(). We will later on in the series build the configuration driver inside a ddConfiguration.dll assembly, together with the other ddFramework classes inside a single namespace. All classes will be defined as mostly singleton and internal. We don’t want it to be called by some careless programming. Our ApplicationDriver have a similar structure to the ConfigurationDriver class. It is internal and sealed with static and hidden constructors and a static Inst property to access the instance of the class running. The only difference is however that it contains two Exec() methods. One Exec() method declared as internal public so that our ConfigurationDriver can execute it, and one hidden that accept an integer parameter very cryptically defined as iApp for no obvious reason (Listing 3). We will look at the Exec methods in more detail.

Listing 3: Application driver class

```
#using jhnFT.Utils.Config
#using System.Windows.Forms
#using System.Collections.Generic

BEGIN NAMESPACE jhnFT.Utils.Config

  INTERNAL SEALED CLASS jhnApplicationDriver
     STATIC HIDDEN _inst AS jhnApplicationDriver

     STATIC CONSTRUCTOR()
        _inst := jhnApplicationDriver{}
     RETURN

     HIDDEN CONSTRUCTOR()
        SUPER()
        SELF:InitializeAppDriver()
     RETURN

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

     HIDDEN METHOD InitializeAppDriver() AS VOID
//      SELF:Exec() //Test if only 1 Exec() can be requested
     RETURN

     METHOD Exec() AS VOID
        STATIC LOCAL iCount := 0 AS INT
        LOCAL kvp AS KeyValuePair<INT, STRING>
        IF iCount++ = 0 // Can only be executed 1 time per application
           kvp := (KeyValuePair<INT, STRING>)SetupDict.Inst:PropertyGet("Start")
           IF kvp:Key > 0
              SELF:Exec(kvp:Key)
           ELSE
              MessageBox.Show(e"No start application specified\n\nIni file\t: " + ;
                 SetupDict.Inst:PropertyGet("FrameworkIni"):ToString() + ;
                                           e"\nSection\t: [system]\nItem\t: class")
           ENDIF
        ELSE
           MessageBox.Show(;
              "Only one instance of the application driver is allowed per active session!", ;
              SELF:GetType():ToString(),MessageBoxButtons.OK, MessageBoxIcon.Stop)
        ENDIF
     RETURN

     HIDDEN METHOD Exec(iApp AS INT) AS VOID
        LOCAL oClsP, oLkpIP AS jhnParameterCollection
        LOCAL ddSD AS SetupDict
        ddSD := SetupDict.Inst
        IF (oClsP := ddSD:ClassPropertyGet(iApp)) = NULL
           MessageBox.Show(
              "Class : " + iApp:ToString() + " does not exist!", SELF:GetType():ToString())
        ELSEIF (oLkpIP := ddSD:LkpItemGet(oClsP:GetInt("classtype_no"))) = NULL
           MessageBox.Show(e"Class_no\t:" + iApp:ToString() + " class type does not exist!",;
                          SELF:GetType():ToString())
        ELSEIF oLkpIP:GetParameter("lkpitem_id") == "application"
           oClsP:AddParameter("classtype_id", oLkpIP:GetParameter("lkpitem_id"))
           BEGIN SCOPE
              LOCAL o AS OBJECT
              o := ddMemberInterface.Inst:MemberAdd(oClsP)
              IF o:GetType():IsSubclassOf(typeof(System.Windows.Forms.Form))
                 Application.Run((Form)o)
              ENDIF
           END SCOPE
        ELSE
           MessageBox.Show("Start class can only be of classtype application", ;
              SELF:GetType():ToString(), MessageBoxButtons.OK, MessageBoxIcon.Stop)
        ENDIF
     RETURN

  END CLASS

END NAMESPACE
```

4.1 The internal public method Exec()

Two things are (immediately) obvious from the method (Listing 4). It contains a STATIC LOCAL iCount variable with an initial value of 0. It also contains a call to a (again)

Singleton class jhnSetupDict getting some property called “Start” that is stored in a KeyValuePair object. More later about the SetupDict class. It then passes the kvp:Key to Exec(iApp). The applicable code is highlighted in blue, the core of the method.

Listing 4: jhnApplicationDriver:Exec()

```
METHOD Exec() AS VOID
   STATIC LOCAL iCount := 0 AS INT
   LOCAL kvp AS KeyValuePair<INT, STRING>
   IF iCount++ = 0 // Can only be executed 1 time per application
      kvp := (KeyValuePair<INT, STRING>)jhnSetupDict.Inst:PropertyGet("Start")
      IF kvp:Key > 0
         SELF:Exec(kvp:Key)
      ELSE
         MessageBox.Show(e"No start application specified\n\nIni file\t: " + ;
            SetupDict.Inst:PropertyGet("FrameworkIni"):ToString() + ;
                                       e"\nSection\t: [start]\nItem\t: class=<value>")
      ENDIF
   ELSE
      MessageBox.Show(;
         "Only one instance of the application driver is allowed per active session!", ;
         SELF:GetType():ToString(),MessageBoxButtons.OK, MessageBoxIcon.Stop)
   ENDIF
RETURN
```

We somehow need to tell our Application driver where to start. Where would be a better place than in our application ini file? It just seem logic that our application will be of a class object. We need to however tell it which class to instantiate. Inspecting our ini file, we find that we define classes in a section class and inexplicably each item has a unique number class_no☺ The obvious is listed in Listing 5.

Listing 5: Start section added to ddFramework.exe.ini

```
[start]
class=1
```

It appears that behind the scenes our jhnSetupDict class is able to get this detail for us and that it is returned in a KeyValuePair<INT, STRING> which seems to tied up with class_no (INT) and class (STRING). The INT appears to be passed on to Exec(iApp).

4.2 The hidden method Exec(iApp)

We call on Columbo to help solve the case (Listing 6). Our application driver has effectively get us to a stage where we can start what seems to be the same as the normal start of an application. We now know the identifier (iApp) of our application and it is time to ask it to get into action. Again we don’t want our application to be started again while it is running, we might burn out the starter motor. A STATIC iCount seems to be the solution again. It seems we need to get some parameters that describe our application and we call it a parameter collection. The jhnSetupDict seems to be quite a clever class, not only can it retrieve a key value pair, put it can also supply class properties [ClassPropertyGet(iApp)] and lookup items [LkpItemGet(classtype_no)]. Looking at our ini file we can hence make the assumption that it supply a single point to request details from the ini, but also some additional key value pairs (Environment variables). In our previous example HelloWorldVN we had instances of the ini all over the show and it seems jhnSetupDict encapsulated it nicely in one place. We also had calls to a memberinterface in our previous sample application, and again it seems it still exists (jhnMemberInterface). The parameters passed however look a bit different, but seems to be available via the parameter collections (jhnParameterCollection).

Listing 6: jhnApplicationDriver:Exec(iApp)

```
HIDDEN METHOD Exec(iApp AS INT) AS VOID
  STATIC LOCAL iCount := 0 AS INT
  LOCAL oClsP, oLkpIP AS jhnParameterCollection
  LOCAL ddSD AS jhnSetupDict
  IF iCount++ = 0
     ddSD := jhnSetupDict.Inst
     IF (oClsP := ddSD:ClassPropertyGet(iApp)) = NULL
        MessageBox.Show("Class : " + iApp:ToString() + " does not exist!", ;
                        SELF:GetType():ToString())
     ELSEIF (oLkpIP := ddSD:LkpItemGet(oClsP:GetInt("classtype_no"))) = NULL
        MessageBox.Show(e"Class_no\t: " + iApp:ToString() + " class type does not exist!", ;
                        SELF:GetType():ToString())
     ELSEIF oLkpIP:GetParameter("lkpitem_id") == "application"
        BEGIN SCOPE
           LOCAL o AS OBJECT
           o := jhnMemberInterface.Inst:MemberAdd(oClsP)
           IF o:GetType():IsSubclassOf(typeof(Form))
              TRY
                 Application.Run((Form)o)
              CATCH ex AS Exception
                 MessageBox.Show(ex:Message, SELF:GetType():ToString() + ":Exec(" + ;
                                 iApp:ToString() + ")")
              END TRY
           ENDIF
        END SCOPE
     ELSE
        MessageBox.Show("Start class can only be of classtype application", ;
                        SELF:GetType():ToString(), ;
                        MessageBoxButtons.OK, MessageBoxIcon.Stop)
     ENDIF
  ELSE
     MessageBox.Show("Only one application driver is allowed per active session!", ;
                     SELF:GetType():ToString(), ;
                     System.Windows.Forms.MessageBoxButtons.OK, ;
                     System.Windows.Forms.MessageBoxIcon.Stop)
  ENDIF
RETURN
```

Unfortunately it seems we were not able to adhere to our 50 lines per class with the Application Driver. On average we still ok though.☺ It is time to look at this new member interface tool and if it can help us to get out average lines of code back to below 50.

5. The class member interface layer

It appears our member interface class still have some similarities to the previous example published. Delegates are still used and it still uses the exact same syntax MemAdd(<OBJECT>). A singleton class is still the order (or is it flavour) of the day. A bit of replicated code though, but we managing below 50 lines per class averages, so we not going to split hairs or in software terms remove some empty lines… At least we able to show the whole class on one printed page (Listing 7), with even a bit of space to write on and do something for the environment, for those like me who prefer to read black on white in hard copy format. Or allow us to write some useless information. I had a big slogan on my wall when I started working: *"If you have nothing to, do don't do it here"*. Or in (data-driven) programming terms: *"If you don't have to programme, don't do it"*. Weather seems quite good for camping and fishing…

Listing 7:The class member interface: jhnMemberInterface

```
#using jhnFT.Utils.Config
#using System.Reflection
DELEGATE MemAdd(o AS OBJECT) AS VOID
BEGIN NAMESPACE jhnFT.Utils.Config
  SEALED CLASS jhnMemberInterface
      STATIC HIDDEN _inst AS jhnMemberInterface
      STATIC CONSTRUCTOR()
         _inst := jhnMemberInterface{}
      RETURN
      HIDDEN CONSTRUCTOR()
         SUPER()
      RETURN
      STATIC PROPERTY Inst AS jhnMemberInterface
         GET
            RETURN _inst
         END GET
      END PROPERTY
      METHOD MemberAdd(oPC AS jhnParameterCollection) AS OBJECT
         LOCAL oAss AS Assembly
         LOCAL o AS OBJECT
         IF oPC:HasKey("defaultclass")
            oAss := Assembly.GetAssembly(SELF:GetType())
            BEGIN SCOPE
               LOCAL ctor AS ConstructorInfo
               LOCAL typ AS Type
               typ := oAss:GetType(oPC:GetParameter("defaultclass"))
               ctor := typ:GetConstructor(<Type>{oPC:GetType()})
               o := ctor:Invoke(<OBJECT>{oPC})
            END SCOPE
         ELSE
            oPC:DisplayMembers("Missing default class")
         ENDIF
      RETURN o
      METHOD MemberAdd(oPC AS jhnParameterCollection, memadd AS MemAdd) AS VOID
         LOCAL o AS OBJECT
         o := SELF:MemberAdd(oPC)
         memadd(o)
      RETURN
  END CLASS
END NAMESPACE
```

Well after all that, we even have some empty space to fill on this page, and I thought I am going to get away with it. We will look at the overloaded MemberAdd methods of the class, since the ApplicationDriver called the single parameter version. A slight variation to what we had in article 5. But it seems it is still receiving some properties describing a member class. However it returns an object and does not make use of a delegate. Will have to stop writing now, the empty space is filled. On to our MemberAdd method receiving one parameter and returning an object.

5.1 jhnMemberInterface:MemberAdd(<parameter collection>)

I we look at the MemberAdd method of Article 5, the one clear observation is that each time we add another type of member, we have work to do to our ever increasing list handled with an IF statement:

```
IF <member>:StartsWith("<membertype>") // The known
  …
[ELSEIF <member>:StartsWith("<membertype>")] // The unknown
  …
ELSE // Even deeper into the unknown
  // Don't know what to do
ENDIF
```

In our first article the statement was made that one of the fundamental issues with Functional Decomposition is that very seldom all the requirements can be gathered before system development. Our IF statement above proofs the point. We need to find a way of addressing the known and unknown requirements. In Clipper, VO and Vulcan we have macro-compiled codeblocks, which is a very under utilised feature. More of that in a future article, however in Visual Objects, we were able to CreateInstance(), Send(), SendClass(), etc. We had a method to speed up the macro-compiled codeblock interface. In .NET we have similar capabilities via the namespace System.Reflection. What we need is to create a string based logic around it. Codeblocks give and still can provide us that capability at a cost of execution speed. VO gave us an improved interface, and .NET although named differently the same [enhanced] features. Our MemberAdd was changed accordingly and it appears we are able to create objects from known and unknown classes, provided they implement a constructor overload that excepts our magical jhnParameterCollection object (Listing 8):

Listing 8: jhnMemberInterface:MemberAdd(oPC)

```
METHOD MemberAdd(oPC AS jhnParameterCollection) AS OBJECT
  LOCAL o AS OBJECT
  IF oPC:HasKey("defaultclass")
     BEGIN SCOPE
        LOCAL oAss AS Assembly
        LOCAL ctor AS ConstructorInfo
        LOCAL typ AS Type
        oAss := Assembly.GetAssembly(SELF:GetType())
        typ := oAss:GetType(oPC:GetParameter("defaultclass"))
        ctor := typ:GetConstructor(<Type>{oPC:GetType()})
        o := ctor:Invoke(<OBJECT>{oPC})
     END SCOPE
  ELSE
     oPC:DisplayMembers("Default class missing - " + ;
                          SELF:GetType():ToString() + ":MemberAdd(<oPC>)")
  ENDIF
RETURN o
```

5.2 jhnMemberInterface:MemberAdd(<params>, <delegate>)

Our MemberAdd method with an additional parameter is a non-event. All it does is accept an object and pass it into the delegate property memadd (Listing 9).

Listing 9: jhnMemberInterface:MemberAdd(oPC, memadd)

```
METHOD MemberAdd(oPC AS jhnParameterCollection, memadd AS MemAdd) AS VOID
  LOCAL o AS OBJECT
  o := SELF:MemberAdd(oPC)
  memadd(o)
RETURN
```

It appears we have mission accomplished. From our ApplicationDriver it seems we have an application going the Start of a normal Form application. Let’s look at our application.

6. The data-driven application start

The ApplicationDriver in essence only try and see if what we supplied in our start section of the ini file is in fact an application and pass the value (iApp) to our MemberInterface. Our MemberInterface try and create an object from the defaultclass property and if successful return it to the ApplicationDriver. It does not really know what it will create, but trust the calling object to know why it requested the object. The ApplicationDriver check if it is an object of type form and would in that case behave like a normal Start function. It will execute an Application.Run(oForm), which again will handle it until the form is closed, or somewhere in the chain of member objects or events of oForm, an Application.Exit() is performed. Since we the all knowing of the known and unknown (no pun intended), we unknowingly know that the ApplicationDriver will receive a jhnApplication object an therefore exit the application. The MemberInterface object will create an object of jhnApplication. Lets look at the application object (Listing 10).

Our application class have some funny properties: nID, Name and Text. If we remember from our article 5 application that should [b]ring a Chr(7) in our minds. It resembles the AppForm class in that application, with the addition of an ID field. Not surprising but the constructor calls an InitializeApp method passing the ParameterCollection. It seems except for our MemberInterface class, the SetupDict class also do some important work as it is visible inside this class too. A new method that we did not see before seems to be part of this nifty class, ClassMemberGet. Well seems we might have plenty of members, since the InitializeApp executes a loop for each member. Not only looping through the members, but our MemberInterface seems to be quite an overworked object, luckily sharing the workload with SetupDict.

But hang on, we have not displayed our Application yet and here we wander off to tell MemberInterface again that we have some members. It is getting confusing, we have not even presented our application and it seem the application is distracted. *"Hey! I want a form object that I can run, anytime soon you will give it to me?"*

Listing 10: The data-driven application class

```
#using jhnFT.Utils.Config
#using System.Windows.Forms
SEALED CLASS jhnApplication
  PROTECT nID AS INT
  PROTECT Name AS STRING
  PROTECT Text AS STRING
  CONSTRUCTOR(oPC AS jhnParameterCollection)
     SUPER()
     SELF:InitializeApp(oPC)
  RETURN
  HIDDEN METHOD InitializeApp(oPC AS jhnParameterCollection) AS VOID
     SELF:nID := oPC:GetInt("class_no")
     SELF:Name := oPC:GetParameter("class_id")
     SELF:Text := oPC:GetParameter("text")
     BEGIN SCOPE
        LOCAL aMbr AS jhnParameterCollection[]
        aMbr := jhnSetupDict.Inst:ClassMemberGet(SELF:nID)
        FOR LOCAL mbr := 0 AS INT UPTO aMbr:Length - 1
           LOCAL o AS OBJECT
           TRY
              Application.Run((Form)(o := jhnMemberInterface.Inst:MemberAdd(aMbr[mbr])))
           CATCH ex AS Exception
              aMbr[mbr]:DisplayMembers(SELF:GetType():ToString() + ":InitializeApp(<oPC>)")
              MessageBox.Show(ex:Message, SELF:GetType():ToString())
           END TRY
        NEXT
     END SCOPE
  RETURN
END CLASS
```

Enough said, we know that application will create an object of type AppForm. Let's look at our application form and stop worrying when it will be returned by or MemberInterface.

7. The data-driven application form

Finally we getting to our application form after about 250 lines of code. That was damn hard work and hope it will be a lot less effort from here onwards. Comparing the code from our application class (Listing 10) to that in Listing 11, it looks very similar, except that a new method was created (ControlsAdd) that by some chance also ask for some members and pass it onto our MemberInterface. Only difference is that it tells MemberInterface to use ControlAdd to associate the members with AppForm.

We can clearly see that our data-driven application framework is building up a pattern. We consistently seems to be starting to use the same concept over and over again. Yes the speed of execution is getting substantially slower, however, if we compare it to requesting a webpage it seems to be still a lot faster. I am sure our users will not even detect the speed penalty.

Listing 11: The data-driven application form

```
#using System.Windows.Forms
#using jhnFT.Utils.Config

CLASS jhnAppForm INHERIT Form
  PROTECT nID AS INT

  CONSTRUCTOR(oPC AS jhnParameterCollection)
     SUPER()
     SELF:InitializeForm(oPC)
  RETURN

  METHOD InitializeForm(oPC AS jhnParameterCollection) AS VOID
     SELF:nID := oPC:GetInt("member_no")
     SELF:Name := oPC:GetParameter("member_id")
     SELF:Text := oPC:GetParameter("text")
     SELF:SuspendLayout()
     SELF:ControlsAdd()
     SELF:ResumeLayout()
  RETURN

  METHOD ControlsAdd() AS VOID
     LOCAL aMbr AS jhnParameterCollection[]
     aMbr := jhnSetupDict.Inst:ClassMemberGet(SELF:nID)
     BEGIN SCOPE
        LOCAL delCtrlAdd AS MemAdd
        delCtrlAdd := MemAdd{SELF, @ControlAdd()}
        FOR LOCAL mbr := 0 AS INT UPTO aMbr:Length - 1
           jhnMemberInterface.Inst:MemberAdd(aMbr[mbr], delCtrlAdd)
        NEXT
     END SCOPE
  RETURN

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

END CLASS
```

It is time to test our statement that we busy building a pattern of how we create and interface to presentation objects of new classes in the known and unknown. Our ini file tells us that AppForm contains a Menu that has members of MenuItem.

8. The data-driven menu

We can see if we can use our AppForm class and with the assistance of our favourite developer tool (NotePad) use a bit of copy, paste and replace to get a menu. Miraculously with a little bit of finetuning, we end up with a menuclass (Listing 12) and a menu item class (Listing 13). It is just another presentation layer and we have discovered in our previous articles that on an abstract layer all presentation layer classes have similar fundamentals. The changes made to the MenuClass (jhnMenu) and MenuItemClass (jhnMenuItem) are highlighted in blue. I think we can give self another pat on the shoulder, we have just created a menu platform for (almost) no additional work in the known and unknown future☺ So much for movies about back to the future. We have grabbed the future and made it happen today!

All we need to do now is explain what we have done to end up with this marvellous piece of art.

Listing 12: The data-driven menu class

```
#using System.Windows.Forms
#using jhnFT.Utils.Config

CLASS jhnMenu INHERIT MenuStrip
  HIDDEN nID AS INT

  CONSTRUCTOR(p AS jhnParameterCollection)
     SUPER()
     SELF:nID := p:GetInt("member_no")
     SELF:Name := p:GetParameter("member_id")
     SELF:Text := p:GetParameter("text")
     SELF:InitializeMenu()
  RETURN

  METHOD InitializeMenu() AS VOID
     LOCAL delMIAdd AS MemAdd
     LOCAL aMbr AS jhnParameterCollection[]
     delMIAdd := MemAdd{SELF, @MenuItemAdd()}
     aMbr := jhnSetupDict.Inst:ClassMemberGet(SELF:nID)
     FOR LOCAL mbr := 0 AS INT UPTO aMbr:Length - 1
        jhnMemberInterface.Inst:MemberAdd(aMbr[mbr], delMIAdd)
     NEXT
  RETURN

  METHOD MenuItemAdd(o AS OBJECT) AS VOID
     IF o:GetType():IsSubclassOf(typeof(ToolStripItem))
        SELF:Items:Add((ToolStripItem)o)
     ENDIF
  RETURN

END CLASS
```

Listing 13: The data-driven menu item class

```
#using System.Windows.Forms
#using jhnFT.Utils.Config

CLASS jhnMenuItem INHERIT ToolStripMenuItem
  HIDDEN nID AS INT

  CONSTRUCTOR(p AS jhnParameterCollection)
     SUPER()
     SELF:nID := p:GetInt("member_no")
     SELF:Name := p:GetParameter("member_id")
     SELF:Text := p:GetParameter("text")
     SELF:InitializeMenuItem()
  RETURN

  METHOD InitializeMenuItem() AS VOID
     LOCAL delMIAdd AS MemAdd
     LOCAL aMbr AS jhnParameterCollection[]
     delMIAdd := MemAdd{SELF, @MenuItemProcess()}
     aMbr := jhnSetupDict.Inst:ClassMemberGet(SELF:nID)
     FOR LOCAL mbr := 0 AS INT UPTO aMbr:Length - 1
        jhnMemberInterface.Inst:MemberAdd(aMbr[mbr], delMIAdd)
     NEXT
  RETURN

  HIDDEN METHOD MenuItemProcess(o AS OBJECT) AS VOID
     IF o:GetType():IsSubclassOf(typeof(ToolStripItem))
        SELF:MenuItemAdd((ToolStripItem)o)
     ENDIF
  RETURN

  HIDDEN METHOD MenuItemAdd(o AS ToolStripItem) AS VOID
     IF o:GetType():IsSubclassOf(typeof(ToolStripMenuItem)) && ;
           !((ToolStripMenuItem)o):HasDropDown
        o:Click +=EventHandler{SELF, @MenuItemClick()}
     ENDIF
     SELF:DropDown:Items:Add(o)
  RETURN

  HIDDEN METHOD MenuItemClick(o AS OBJECT, e AS EventArgs) AS VOID
     MessageBox.Show(((ToolStripMenuItem)o):Text:Replace("&", ""))
  RETURN

  HIDDEN METHOD MenuClose(o AS OBJECT, e AS EventArgs) AS VOID
     MessageBox.Show("Thank you for using the application”+ ;
                     “\nHope to see you soon again\n\n" +;
                     SELF:Text:Replace("&", ""))
     Application.Exit()
  RETURN

END CLASS
```

WOW, that was some hard work. We in essence changed our ControlAdd on AppForm to MenuItemAdd. It did take some thinking, since it seems there are different ways how our MenuItems can behave. On MenuClass we add items to Items, and MenuItems that are members of other MenuItems is added to DropDown:Items. If this MenuItem added does not contain other members in its DropDown, we associate a MenuClickEvent to it.

Lets see if our copy, paste, replace and delete theory is holding up to a data-driven SeparatorClass (Listing 14).

Listing 14: The data-driven toolstrip separator class

```
#using System.Windows.Forms

CLASS jhnToolStripSeparator INHERIT ToolStripSeparator
  HIDDEN nID AS INT
  CONSTRUCTOR(p AS jhnParameterCollection)
     SUPER()
     SELF:nID := p:GetInt("classmember_no")
     SELF:Name := p:GetParameter("class_id")
  RETURN
END CLASS
```

And with a legal hat on, I will end this with *"Your honour, with that I conclude my case that the functional decomposition methodology is drowning the software development industry in their own information pollution"*.

There are two classes that we did not address yet, but we made reference to them on numerous occasions: ParameterCollection and SetupDict.

9. The data-driven parameter collection class

The jhnParameterCollection class is a wrapper around System.Collections.Specialized.NameValueCollection. We probably could have used a simple SortedList<STRING, STRING>, however for future use the NameValueCollection was chosen. It works on a very similar principle as SortedList, however when it finds a duplicate key it appends the value to the Value as a comma separated string and does not throw an exception, hence the ValueCollection part of the classname. The class was enhanced a bit to extend the Get() method to include GetInt(), GetWord(), GetDWord(), GetLong(), GetReal4(), GetReal8(), GetDecimal() and GetBoolean().

It was also enhanced to be passed an EAV string and parse it into KeyValue pairs that are added to the Collection. For debugging purposes a DisplayMember() method was added to show all the Key and ValueCollections associated with it. And a HasKey() to see if a certain key already exists. The Set() method was also encapsulated by a Put() method with the same parameters to allow the "old" value to be returned. I excluded all the Get<Type>() methods in the article to save on paper (Listing 15).

Listing 15: The data-driven parameter collection class

```
#using System
#using System.Windows.Forms
CLASS jhnParameterCollection INHERIT System.Collections.Specialized.NameValueCollection
  CONSTRUCTOR()
     SUPER()
  RETURN
  METHOD AddFromEAVString(cStr AS STRING) AS VOID
     LOCAL sSplit, sSub AS STRING[]
     LOCAL nCnt AS INT
     sSplit := cStr:Split(";":ToCharArray(), StringSplitOptions.RemoveEmptyEntries)
     FOR nCnt := 0 UPTO sSplit:Length - 1
        IF sSplit[nCnt]:Contains("=")
           sSub := sSplit[nCnt]:Split("=":ToCharArray())
           TRY
              SELF:Add(sSub[0]:ToLower(), ;
                       sSub[1]:Replace("$eq$", "="):Replace(“$sc$, “;”))
           CATCH oEx AS Exception
              MessageBox.Show(cStr + "(" + nCnt:ToString() + ")" + e"\n" + oEx:Message, ;
                              SELF:GetType():ToString() + “:AddFromEAVString(<string>)”)
              SELF:DisplayMembers(SELF:GetType():ToString() + “:AddFromEAVString(<string>)”)
           END TRY
        ENDIF
     NEXT
  RETURN
  METHOD GetInt(strName AS STRING) AS INT
     LOCAL nRet AS INT
     nRet := Int16.Parse(SELF:Get(strName))
  RETURN nRet
  METHOD GetBoolean(strName AS STRING) AS LOGIC
     LOCAL lBoolean AS LOGIC
     LOCAL strRet := SELF:Get(strName):Replace(" ", "") AS STRING
     IF strRet:Length > 0
         lBoolean := "_t_true_y_yes_1_":Contains("_" + strRet + "_")
     ENDIF
  RETURN lBoolean
  METHOD HasKey(cKey AS STRING) AS LOGIC
  RETURN Array.IndexOf(SELF:AllKeys, cKey) >= 0
  METHOD Put(strName AS STRING, strVal AS STRING) AS STRING
     LOCAL sOld AS STRING
     strName := strName:ToLower()
     sOld := SELF:Get(strName)
     IF sOld = NULL
        sOld := ""
     ENDIF
     SELF:Set(strName, strVal)
  RETURN sOld
  METHOD DisplayMembers() AS VOID
     SELF:DisplayMembers(ProcName(1) + "(" + ProcLine(1):ToString() + ")")
  RETURN
  METHOD DisplayMembers(cTxt AS STRING) AS VOID
     LOCAL cStr AS STRING
     LOCAL nItem AS INT
     cStr := ""
     FOR nItem := 0 TO SELF:Count - 1
        cStr += SELF:GetKey(nItem) + ":" + SELF:Get(nItem) + e"\n"
     NEXT
     MessageBox.Show(cStr, cTxt)
  RETURN
END CLASS
```

10. The data-driven setup dictionary class

I am not going into all the details of this class since it will change a bit later on in the series. I will basically highlight the important parts from a data-driven perspective. The SetupDict is in principle a class that is used as a “global” store property collection, call it gloabal application vars container of what we need to run our application(s). Firstly, it setup some properties, e.g. ExeName, StartupPath, etc and have a method for accessing them PropertyGet().

It also reads our ini file and store it in structures to enable fast retrieval of information e.g LkpItemGet(), ClassPropertyGet() and ClassMemberGet(). It will contain some delegates that we can use later on in our application, too much information at the moment.

I conclude: “Data-driven applications are a fun way in which to develop software”. I hope you enjoyed the reading and that it gave you some ideas that can be implemented in your own environment.

11. Summary

We have created a framework of classes that can be reused over and over again in a consistent way. It was based on the known, with some anticipation of what are still in the unknown. Happy reading till the next article!