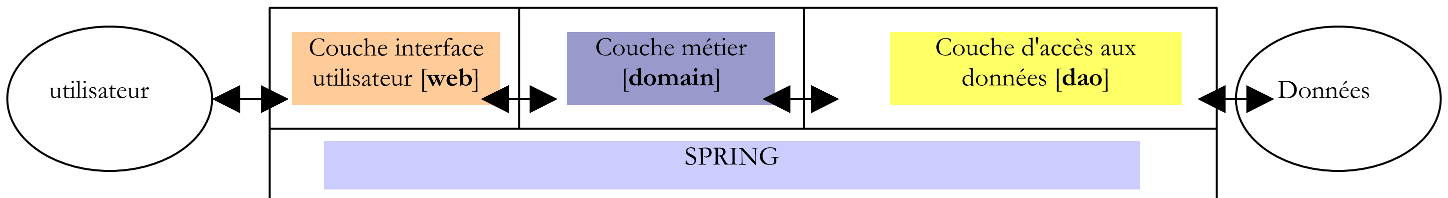
M2VC-win, un moteur MVC pour des applications windows .NET

serge.tahe@istia.univ-angers.fr, juin 2005

1 Introduction

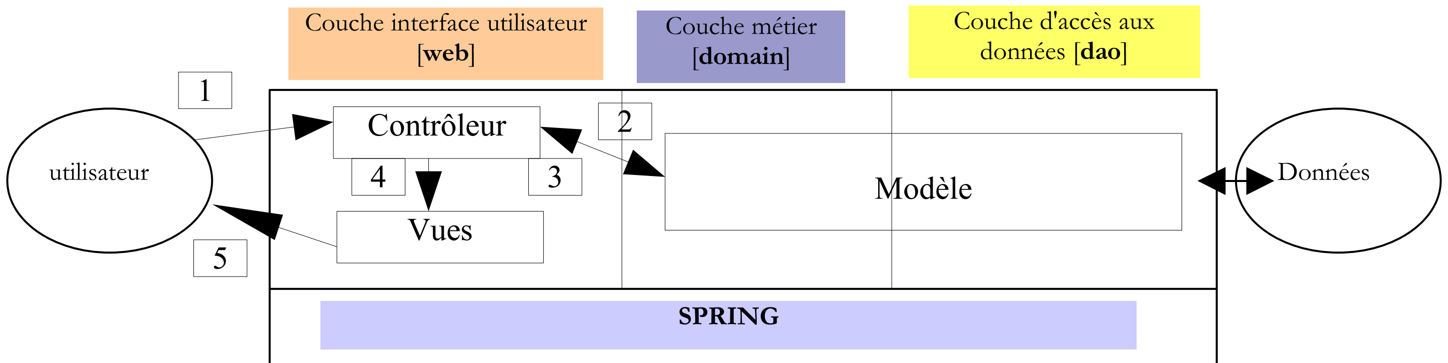
Nous nous proposons ici d'étudier un moteur MVC pour des application windows développées sous .NET. Si le modèle MVC (Modèle - Vue - Contrôleur) est désormais bien accepté dans le cadre des applications web, il ne semble pas qu'il ait percé dans le développement d'applications windows ou alors on n'en parle pas. L'idée de ce moteur MVC est venu à l'occasion du portage d'une interface web existante vers une interface à base de formulaires windows. L'interface web ayant une architecture MVC, j'ai souhaité reproduire celle-ci dans l'interface windows. En l'absence d'outils connus, j'ai été amené à développer M2VC-win.

Considérons l'application web à trois couches suivante :



- les trois couches sont indépendantes grâce à l'utilisation d'interfaces
- l'intégration des différentes couches est réalisée avec **Spring**

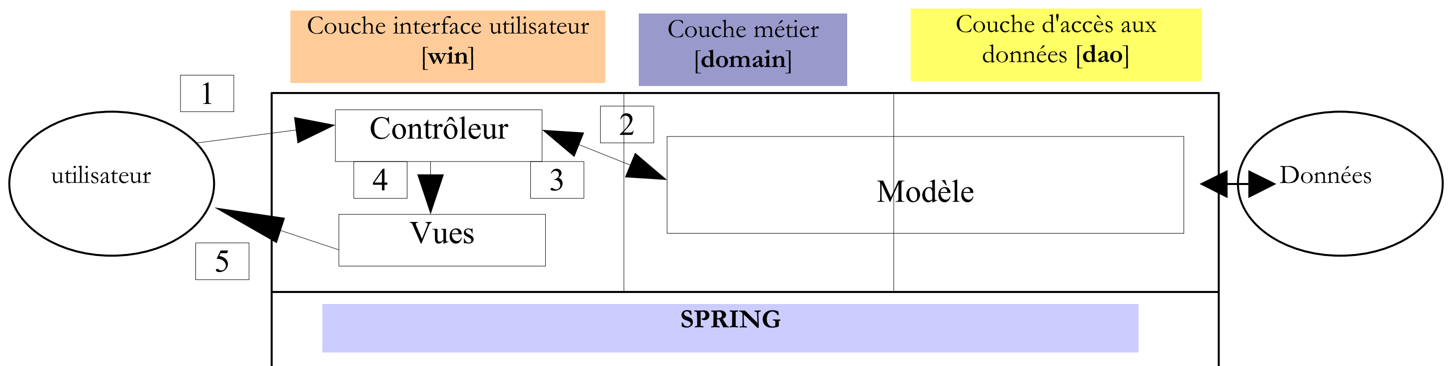
L'architecture MVC (Modèle - Vue - Contrôleur) s'intègre dans le schéma en couches ci-dessus, de la façon suivante :



Le traitement d'une demande d'un client se déroule ainsi :

1. le client fait une demande au contrôleur. Celui-ci voit passer toutes les demandes des clients. C'est la porte d'entrée de l'application. C'est le C de MVC.
2. le contrôleur traite cette demande. Pour ce faire, il peut avoir besoin de l'aide de la couche métier, ce qu'on appelle le modèle M dans la structure MVC.
3. le contrôleur reçoit une réponse de la couche métier. La demande du client a été traitée. Celle-ci peut appeler plusieurs réponses possibles. Un exemple classique est
   - une page d'erreurs si la demande n'a pu être traitée correctement
   - une page de confirmation sinon
4. le contrôleur choisit la réponse (= vue) à envoyer au client. Celle-ci est le plus souvent une page contenant des éléments dynamiques. Le contrôleur fournit ceux-ci à la vue.
5. la vue est envoyée au client. C'est le V de MVC.

L'architecture précédente transposée dans le monde des applications windows donne la chose suivante :



La couche [**web**] qui présentait des pages web à l'utilisateur est remplacée ici par une couche [**win**] qui lui, présente des formulaires windows. Le fonctionnement de l'ensemble, décrit plus haut, peut être repris ici à l'identique.

Différents outils du monde libre sont disponibles pour faciliter le développement de la couche [web] selon le concept MVC. Les plus connus sont dans le monde Java. On trouve notamment **Struts** et **Spring**. Cet article se propose de construire un outil inspiré de **Struts** pour faciliter l'implémentation du concept MVC dans la couche [win] de l'application à trois couches ci-dessus. Par la suite, nous appellerons cet outil, Moteur MVC pour windows et le désignerons par **M2VC-win**. Il sera développé en VB.NET.

Avertissement

M2VC-win vise seulement à apporter une contribution à la recherche de méthodologies MVC dans le monde des applications windows. En aucun cas, il ne vise à être la solution définitive. Il n'est ainsi même pas sûr que le modèle utilisé pour son développement soit correct. Les quelques applications que j'ai pu développer avec, et dont on trouvera des exemples dans cet article, sont trop peu nombreuses pour le certifier. Cette contribution peut néanmoins donner des idées à d'autres développeurs.

En-dehors de tout but pratique, M2VC-win est un bon exemple des possibilités apportées par [Spring Ioc].

Pré-requis et outils

Les outils utilisés dans cet articles sont les suivants :

- **Visual Studio.net** pour le développement des applications
- **Spring IoC** pour l'instanciation des objets nécessaires au moteur MVC

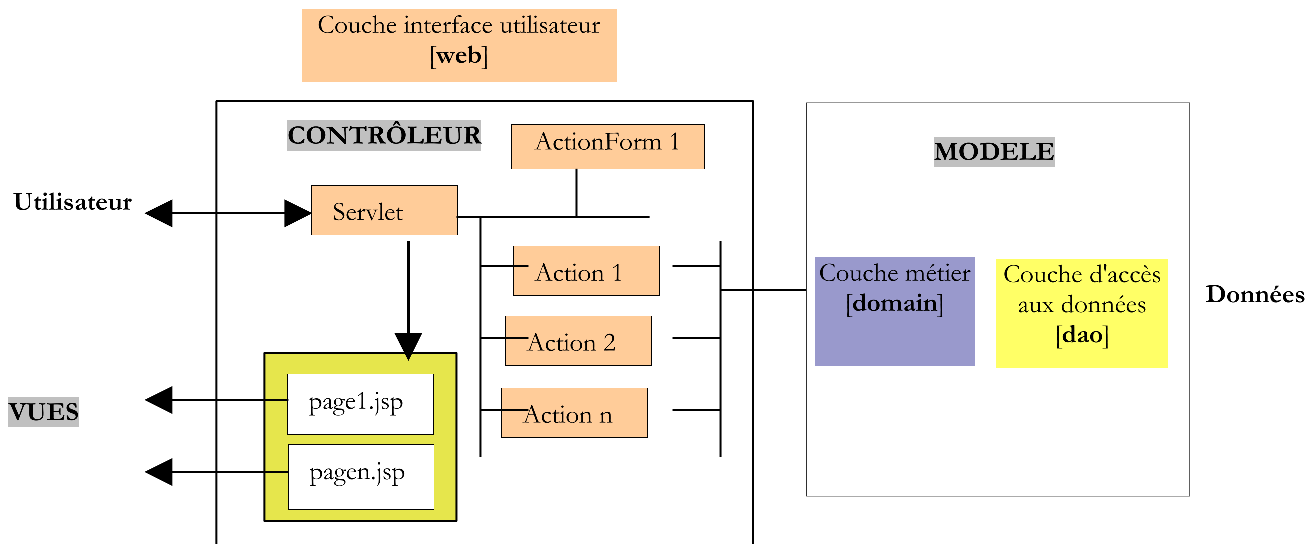
Dans une échelle [débutant-intermédiaire-avancé], ce document est dans la partie [avancé]. Sa compréhension nécessite divers pré-requis qu'on pourra trouver dans certains des documents que j'ai écrits :

- **langage VB.net** : [http://tahe.developpez.com/dotnet/vbnet/] : classes, interfaces, héritage, polymorphisme, threads, synchronisation
- **Spring IoC** : [http://tahe.developpez.com/dotnet/springioc]
- **Struts** : [http://tahe.developpez.com/java/struts/]. Ce pré-requis n'est pas indispensable. La connaissance de Struts facilite simplement la compréhension de l'article.

Il est bien évident que le lecteur peut utiliser ses documents favoris.

2 La philosophie de Struts

Rappelons ou découvrons l'architecture MVC générique utilisée par STRUTS dans le cadre des applications web :

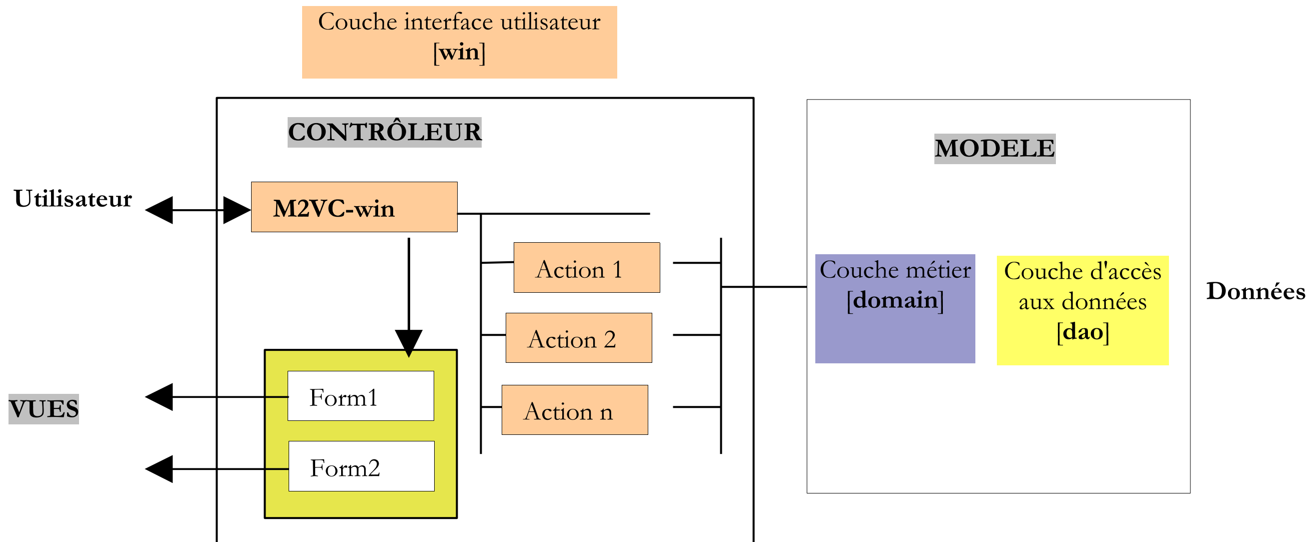


M=modèle	les classes métier, les classes d'accès aux données et la source de données
V=vues	les pages JSP
C=contrôleur	la servlet de traitement des requêtes clientes, les objets [Action] et [ActionForm]

- le contrôleur est le coeur de l'application. Toutes les demandes du client transitent par lui. C'est une servlet générique [ActionServlet] fournie par STRUTS. On peut dans certains cas être amené à la dériver. Pour les cas simples, ce n'est pas nécessaire. Cette servlet générique prend les informations dont elle a besoin dans un fichier le plus souvent appelé **struts-config.xml**.
- la demande du client vient par l'intermédiaire d'une requête HTTP. L'URL cible est l'action demandée par le client.
- si la requête du client contient des paramètres de formulaire, ceux-ci sont mis par le contrôleur dans un objet **[ActionForm]**.
- dans le fichier de configuration **struts-config.xml**, à chaque URL (=action) devant être traitée par programme (ne correspondant donc pas à une vue JSP qu'on pourrait demander directement) on associe certaines informations :
    - le **nom de la classe** de type **Action** chargée d'exécuter l'action
    - si l'URL demandée est paramétrée (cas de l'envoi d'un formulaire au contrôleur), le nom de l'objet [**ActionForm]** chargé de mémoriser les informations du formulaire
- muni de ces informations fournies par son fichier de configuration, à la réception d'une demande d'URL par un client, le contrôleur est capable de déterminer s'il y a un objet [**ActionForm]** à créer et lequel. Une fois instancié, cet objet [**ActionForm]** peut vérifier que les données qu'on vient de lui injecter et qui proviennent du formulaire, sont valides ou non. Une méthode de [**ActionForm]** appelée **validate** est appelée automatiquement par le contrôleur. L'objet [**ActionForm]** étant construit par le développeur, celui-ci a mis dans la méthode **validate** le code vérifiant la validité des données du formulaire. Si les données se révèlent invalides, le contrôleur n'ira pas plus loin. Il passera la main à une vue dont il trouvera le nom dans son fichier de configuration. L'échange est alors terminé.
- si les données de l'objet [**ActionForm]** sont correctes, ou s'il n'y a pas de vérification ou s'il n'y a pas d'objet [**ActionForm]**, le contrôleur passe la main à l'objet de type [**Action]** associé à l'URL. Il le fait en demandant l'exécution de la méthode **execute** de cet objet à laquelle il transmet la référence de l'objet **[ActionForm]** qu'il a éventuellement construit. Les objets [**Action**] sont construits par le développeur. C'est là qu'il place le code chargé d'exécuter l'action demandée. Ce code peut nécessiter l'utilisation de la couche métier ou modèle dans la terminologie MVC. Les objets [**Action**] sont les seuls objets en contact avec cette couche. A la fin du traitement, l'objet [**Action]** rend au contrôleur une chaîne de caractères représentant le résultat de l'action.
- dans son fichier de configuration, le contrôleur trouvera l'URL de la vue associée à cette chaîne. Il envoie alors cette vue au client. L'échange avec le client est terminé.

3 La philosophie de M2VC-win

Nous nous inspirons ici librement de la philosophie Struts pour construire notre moteur MVC. L'architecture d'une application **M2VC-win** sera la suivante :

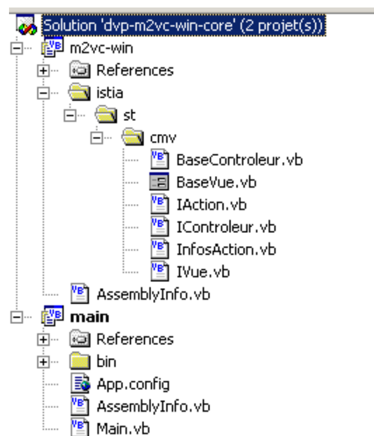


M=modèle	les classes métier, les classes d'accès aux données et la base de données
V=vues	les formulaires Windows
C=contrôleur	le moteur [MVC2-win] de traitement des requêtes clientes, les objets [Action]

Notre moteur **M2VC-win** joue le rôle de la servlet générique [**ActionServlet**] de Struts. **M2VC-win** n'utilisera pas d'objets [**ActionForm**] comme objets tampon entre le client et les classes [**Action**]. Il fera exécuter les actions demandées par l'utilisateur par des objets [**Action**] qui iront chercher les paramètres de la demande du client dans un objet qui sera défini par le développeur. Le moteur ne fait aucune hypothèse sur ce point.

4 Les éléments de M2VC-win

M2VC-win a été construit avec Visual Studio. Le plus simple est peut-être de partir de la structure de la solution VS utilisée pour construire [M2VC-win] puis de détailler un à un ses différents éléments :



La solution VS est composée de deux projets :

- un projet [m2vc-win] de type [Bibliothèque de classes] qui donne naissance à [m2vc-win.dll], la DLL que les projets reposant sur le moteur M2VC-win devront référencer.
- un projet [main] de type [Console] qui donne naissance à l'exécutable [m2vc-win.exe]. C'est le lanceur du contrôleur. Il instancie celui-ci à l'aide d'un fichier de configuration [m2vc-win.exe.config]. Cette instanciation est faite à l'aide de [Spring].

Un projet P utilisant le moteur [M2VC-win] sera un projet de type [Bibliothèque de classes]. Appelons [appli.dll] la DLL du projet P. Une configuration d'exécution typique du projet P sera la suivante :

Nom	Taille
appli.dll	24 Ko
log4net.dll	192 Ko
m2vc-win.dll	13 Ko
m2vc-win.exe	8 Ko
m2vc-win.exe.config	3 Ko
Spring.Core.dll	272 Ko

`m2vc-win.dll`	le moteur M2VC-win
`m2vc-win.exe`	le lanceur de l'application. Il instancie le contrôleur à l'aide du fichier de configuration [m2vc-win.exe.config] et des bibliothèques [Spring]
`Spring.Core.dll, log4net.dll`	les bibliothèques [Spring]
`appli.dll`	la DLL du projet P

Le projet [m2vc-win] de type [Bibliothèque de classes] est formé des éléments suivants :

- trois interfaces : **IAction**, **IVue**, **IControleur**
- deux classes dérivables : **BaseControleur**, **BaseVue**
- une classe : **InfosAction**

Passons en revue le rôle de ces différents éléments sans encore entrer dans les détails. Les explications ci-dessous doivent être lues à la lumière de l'architecture présentée dans le paragraphe précédent :

`IControleur`	dit ce que doit savoir faire un contrôleur générique
`BaseControleur`	implémente l'interface [**IControleur]**. Assure toute la synchronisation [traitement actions] - affichage vues. Le développeur peut dériver cette classe si le contrôleur générique ne lui suffit pas.
`Iaction`	dit ce que doit savoir faire un objet [**Action**]. Aucune implémentation n'est proposée.
`Ivue`	dit ce que doit savoir faire une vue
`BaseVue`	implémente l'interface [**Ivue]** précédente. Est dérivée de l'objet [Form] de .NET. Les formulaires d'une application windows utilisant [M2VC-win] seront dérivés de cette classe de base.
`InfosAction`	classe qui définit les informations liées à une action. C'est grâce à cette classe, que le contrôleur générique va savoir ce qu'il doit faire.

Le projet [main] de type [Console] a deux éléments principaux :

`Main.vb`	va instancier et lancer le contrôleur [m2vc-win]
`App.config`	fichier de configuration du contrôleur [m2vc-win]

Nous décrivons maintenant ces éléments un à un. Ils sont tous placés dans l'espace de noms [istia.st.cmv], où [cmv] signifie " Contrôleur Multi-Vues " qui était le nom initial du projet.

4.1 L'interface [IControleur]

Son code est le suivant :

```
Namespace istia.st.cmv
  Public Interface IControleur
    Sub run()
  End Interface
End Namespace
```

On ne demandera qu'une chose à un contrôleur, c'est de s'exécuter. C'est la méthode [**run**] qui nous permettra de lancer le contrôleur de l'application windows.

4.2 L'interface [IAction]

Son code est le suivant :

```
Namespace istia.st.cmv

  Public Interface IAction
    Function execute() As String
  End Interface

End Namespace
```

On ne demandera qu'une chose à un objet [Action], c'est d'exécuter l'action pour laquelle il a été construit. Rappelons ici que c'est le développeur qui fournira les classes [Action] implémentant l'interface [IAction]. Pour exécuter une action, on appellera donc la méthode [**execute**] de l'objet [Action] associée à cette action.

4.3 L'interface [IVue]

Son code est le suivant :

```
Namespace istia.st.cmv
  Public Interface IVue
    Sub affiche()
    Sub cache()
    Property action() As String
    Property nom() As String
  End Interface
End Namespace
```

Que peut-on demander à une vue ?

- qu'elle s'affiche (méthode **affiche**)
- qu'elle se cache (méthode **cache**)
- qu'elle donne son nom (propriété **nom**). Cette propriété n'a jamais paru indispensable mais elle a été néanmoins conservée. Je l'ai utilisée dans des phases de débogage
- qu'elle donne le nom de l'action demandée par l'utilisateur (propriété **action**). Rappelons qu'une vue sera une fenêtre windows. Celle-ci va réagir toute seule à certains événements. Ce sera la grande différence avec les vues web. Elle peut ainsi réagir à un événement *drag'n drop*. Elle s'interdira de faire appel elle-même aux couches métier dont elle n'est pas sensée avoir connaissance.. Dans ce cas, elle passera plutôt la main au contrôleur en lui donnant le nom de l'action demandée par l'utilisateur.

4.4 La classe [BaseVue]

[**BaseVue**] est une classe de base implémentant l'interface **Ivue**. Outre qu'elle implémente les caractéristiques de l'interface [IVue], elle sait également se synchroniser avec le contrôleur générique. En effet, lorsque celui-ci demande l'affichage d'une vue, il va :

- faire appel à la méthode [affiche] de la vue
- attendre que celle-ci envoie le nom d'une action à exécuter. Il y a donc une synchronisation [controleur - vue] à organiser.

Le code de la classe [BaseVue] est le suivant :

```
1. Imports System.Threading
2. Imports System.Windows.Forms
3.
4. Namespace istia.st.cmv
5.  Public Class BaseVue
6.    Inherits System.Windows.Forms.Form
7.    Implements IVue
8.
9.    ' nom de la vue
10.   Private _nom As String
11.
12.   Public Property nom() As String Implements IVue.nom
13.     Get
14.       Return _nom
15.     End Get
16.     Set(ByVal Value As String)
17.       _nom = Value
18.     End Set
19.   End Property
20.
21.   ' action à exécuter par le contrôleur
22.   Private _action As String
23.
24.   ' propriété associée
25.   Public Property action() As String Implements IVue.action
26.     Get
27.       Return _action
28.     End Get
29.     Set(ByVal Value As String)
30.       _action = Value
31.     End Set
32.   End Property
33.
34.   ' synchro avec le contrôleur
35.   Private _synchro As AutoResetEvent
```

```
36.
37.   ' propriété associée
38.   Public Property synchro() As AutoResetEvent
39.     Get
40.       Return _synchro
41.     End Get
42.     Set(ByVal Value As AutoResetEvent)
43.       _synchro = Value
44.     End Set
45.   End Property
46.
47.   ' variables locales de la vue
48.   Private threadVue As Thread
49.
50.#Region " Code généré par le Concepteur Windows Form "
51.
52.   Public Sub New()
53.     MyBase.New()
54.
55.     'Cet appel est requis par le Concepteur Windows Form.
56.     InitializeComponent()
57.
58.     'Ajoutez une initialisation quelconque après l'appel InitializeComponent()
59.
60.   End Sub
61.
62.   'La méthode substituée Dispose du formulaire pour nettoyer la liste des composants.
63.   Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
64.     If disposing Then
65.       If Not (components Is Nothing) Then
66.         components.Dispose()
67.       End If
68.     End If
69.     MyBase.Dispose(disposing)
70.   End Sub
71.
72.   'Requis par le Concepteur Windows Form
73.   Private components As System.ComponentModel.IContainer
74.
75.   'REMARQUE : la procédure suivante est requise par le Concepteur Windows Form
76.   'Elle peut être modifiée en utilisant le Concepteur Windows Form.
77.   'Ne la modifiez pas en utilisant l'éditeur de code.
78.   <System.Diagnostics.DebuggerStepThrough()> Private Sub InitializeComponent()
79.     '
80.     'AbstractVue
81.     '
82.     Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
83.     Me.ClientSize = New System.Drawing.Size(292, 46)
84.     Me.ControlBox = False
85.     Me.Name = "AbstractVue"
86.     Me.Text = "AbstractVue"
87.
88.   End Sub
89.
90.#End Region
91.
92.
93.   Public Overridable Sub affiche() Implements IVue.affiche
94.     ' on s'affiche dans le thread d'affichage
95.     If threadVue Is Nothing Then
96.       threadVue = New Thread(New ThreadStart(AddressOf run))
97.       threadVue.Start()
98.     Else
99.       ' on s'affiche
100.        Show()
101.      End If
102.    End Sub
103.
104.    Public Overridable Sub cache() Implements IVue.cache
105.      ' on cache le formulaire
106.      Hide()
107.    End Sub
108.
109.    Protected Overridable Sub exécute(ByVal action As String)
110.      ' on note l'action à exécuter
111.      Me.action = action
112.      ' on rend la main au contrôleur
113.      synchro.Set()
114.    End Sub
115.
116.    ' thread d'affichage de la vue
117.    Private Sub run()
118.      ' exécution evts vue
119.      Application.Run(Me)
120.    End Sub
121.
122.    ' demande de fermeture fenêtre
```

```
123.   Private Sub BaseVue_Closing(ByVal sender As Object, ByVal e As
   System.ComponentModel.CancelEventArgs) Handles MyBase.Closing
124.     ' on interdit la fermeture de la fenêtre
125.     e.Cancel = True
126.   End Sub
127. End Class
128.End Namespace
```

Les points à comprendre sont les suivants :

- la classe [BaseVue] dérive de [System.Windows.Forms.Form] (ligne 6). On a généré la classe, avec Visual Studio, comme étant un formulaire windows, ce qui explique la région de définition du formulaire trouvée aux lignes 50 à 90. Cette région a été laissée telle que VS l'a générée. Aucune modification n'y a été apportée. Nous ne la commenterons pas plus.
- la classe implémente l'interface [IVue] - ligne 7
- elle implémente la propriété [nom] de l'interface [Ivue] - lignes 9-19
- elle implémente la propriété [action] de l'interface [Ivue] - lignes 21-32
- elle implémente la méthode [affiche] de l'interface [Ivue] - lignes 93-102
- elle implémente la propriété [cache] de l'interface [Ivue] - lignes 104-107
- elle a un objet de synchronisation [synchro] de type [AutoResetEvent] - lignes 35-45. Cette classe est définie dans l'espace de noms [System.Threading] d'où l'importation de cet espace de noms en ligne 1. La propriété [synchro] va permettre à la vue de signaler au contrôleur qu'elle demande l'exécution d'une action.
- le formulaire windows va être affiché dans un thread différent de celui dans lequel s'exécute le contrôleur. Ce thread est déclaré en ligne 48

- **la méthode [affiche]**

```
129.   Public Overridable Sub affiche() Implements IVue.affiche
130.     ' on s'affiche dans le thread d'affichage
131.     If threadVue Is Nothing Then
132.       threadVue = New Thread(New ThreadStart(AddressOf run))
133.       threadVue.Start()
134.     Else
135.       ' on s'affiche
136.       Show()
137.     End If
138.   End Sub
139.
140.   Public Overridable Sub cache() Implements IVue.cache
141.     ' on cache le formulaire
142.     Hide()
143.   End Sub
144.
145.   Protected Overridable Sub exécute(ByVal action As String)
146.     ' on note l'action à exécuter
147.     Me.action = action
148.     ' on rend la main au contrôleur
149.     synchro.Set()
150.   End Sub
151.
152.   ' thread d'affichage de la vue
153.   Private Sub run()
154.     ' exécution evts vue
155.     Application.Run(Me)
156.   End Sub
```

La méthode [affiche] est déclarée [Overridable]. Elle est destinée à être spécialisée par des classes dérivées. En effet, [BaseVue] ne sait pas quoi afficher. Seules les classes dérivées le sauront. Elles procèderont à l'affichage du formulaire de la façon suivante :

- initialisation des composants du formulaire
- appel de la méthode [affiche] de la classe de base [BaseVue] ci-dessus. C'est elle qui doit terminer l'affichage. Que fait-elle ?

- ✗ elle regarde si le formulaire a déjà été affiché. Si c'est le cas, le thread d'affichage [threadVue] a été initialisé - ligne 131
- ✗ si le thread d'affichage n'existe pas, alors on le crée (ligne 132) et on le lance (ligne 133). Le thread a été lancé pour exécuter la méthode privée [run] (ligne 132). La méthode [run] s'exécute donc. Que fait-elle ? Elle affiche le formulaire [Me] et lance l'écoute des événements (ligne 155). La méthode [affiche] est terminée.
- ✗ si le thread d'affichage existe déjà, alors on se contente d'afficher la vue (ligne 136). La méthode [affiche] est terminée.
- ✗ dans quelle situation sommes-nous lorsque la méthode [affiche] a été entièrement exécutée ?
  - ✗ l'utilisateur voit le formulaire. Il peut agir dessus. Le formulaire réagit aux événements.
  - ✗ celui-ci est affiché dans un thread [threadVue] séparé du thread du contrôleur
  - ✗ le contrôleur est en attente d'un événement : que la vue lui dise qu'une action lui est demandée. Elle signalera cet événement grâce à la variable de synchronisation [synchro] qui sera partagée entre le contrôleur et toutes les vues de l'application. Pour attendre, le contrôleur a émis la séquence d'instruction suivante :

```
      ' on affiche la vue en se synchronisant sur elle
```

```
      synchro.Reset()
      vue.affiche()
      synchro.WaitOne()
```

- ✗ la vue débloquera le contrôleur en faisant :

```
      synchro.Set()
```

✗ quand la situation se débloque-t-elle ? L'utilisateur interagit avec le formulaire. Celui-ci réagit avec les gestionnaires d'événements qu'on lui a mis. Pour certains événements, le développeur de la vue décide de passer la main au contrôleur. Pour cela il appelle la méthode [execute] de la classe de base [BaseVue] en lui passant le nom de l'action à exécuter.

- **la méthode [execute]**

Son code est le suivant :

```
157.   Protected Overridable Sub exécute(ByVal action As String)
158.     ' on note l'action à exécuter
159.     Me.action = action
160.     ' on rend la main au contrôleur
161.     synchro.Set()
162.   End Sub
```

- ✗ la méthode est protégée [protected]. C'est pour cette raison que les classes dérivées la connaissent.
- ✗ l'action est notée dans la propriété publique [action] - ligne 159. C'est ici que le contrôleur la récupèrera.
- ✗ on débloque le contrôleur - ligne 161. Celui-ci va pouvoir exécuter l'action qu'on lui demande
- ✗ on notera et **c'est très important de le comprendre** qu'en ligne 162, la méthode [execute] est terminée. Son exécution sera demandée à la suite d'un événement du formulaire affiché. La fin de [exécute] entraîne la fin de la gestion de l'événement. De nouveaux événements du formulaire peuvent alors survenir et être traités. **Cela signifie qu'alors que le contrôleur est en train d'accomplir une action pour le compte de la vue, celle-ci continue à répondre aux événements qui la concernent**. Nous n'avons pas voulu prendre de décision sur ce point. C'est au développeur de le faire. **Celui-ci doit savoir qu'à chaque fois qu'il fait appel au contrôleur, il déclenche une action asynchrone**. A lui de savoir comment son formulaire doit se comporter. Dans les exemples étudiés plus loin, nous proposons plusieurs réponses simples à cette situation.

- **la méthode [cache]**

Le contrôleur gère plusieurs vues. Après avoir exécuté l'action demandée par une vue, il peut vouloir afficher une autre vue. Il demande alors à la vue précédente de se cacher. Le code est simple :

```
163.   Public Overridable Sub cache() Implements IVue.cache
164.     ' on cache le formulaire
165.     Hide()
166.   End Sub
```

- **l'événement [BaseVue_Closing]**

L'utilisateur a devant lui une fenêtre qu'il pourrait être tenté de fermer. Nous décidons de le lui interdire. Il devra passer forcément par une option de menu ou un bouton. C'est un choix qui a été pris après des tests pratiqués avec le moteur MVC. Lorsque le contrôleur affiche la fenêtre, il attend que celle-ci lui signale un événement. Si par erreur, l'utilisateur ferme la fenêtre, en ne cliquant pas au bon endroit par exemple, le contrôleur est bloqué en attente d'un événement qui ne viendra plus.

Ce choix d'architecture n'est pas contraignant pour le développeur qui peut définir son propre gestionnaire de l'événement [Closing] comme il le fait d'habitude pour ses formulaires windows. Comme c'est un événement qui assez souvent n'est pas géré (on s'attend à ce que l'application s'arrête sans rien dire), le comportement par défaut amené par [BaseVue] sera exécuté et la fenêtre ne sera pas fermée.

```
167.   ' demande de fermeture fenêtre
168.   Private Sub BaseVue_Closing(ByVal sender As Object, ByVal e As
  System.ComponentModel.CancelEventArgs) Handles MyBase.Closing
169.     ' on interdit la fermeture de la fenêtre
170.     e.Cancel = True
171.   End Sub
172. End Class
```

Les vues de l'application windows, dérivées de [BaseVue], seront instanciées par configuration avec [Spring IoC].

4.5 La classe [InfosAction]

La classe [InfosAction] spécifie une action à exécuter par le contrôleur. Son code est le suivant :

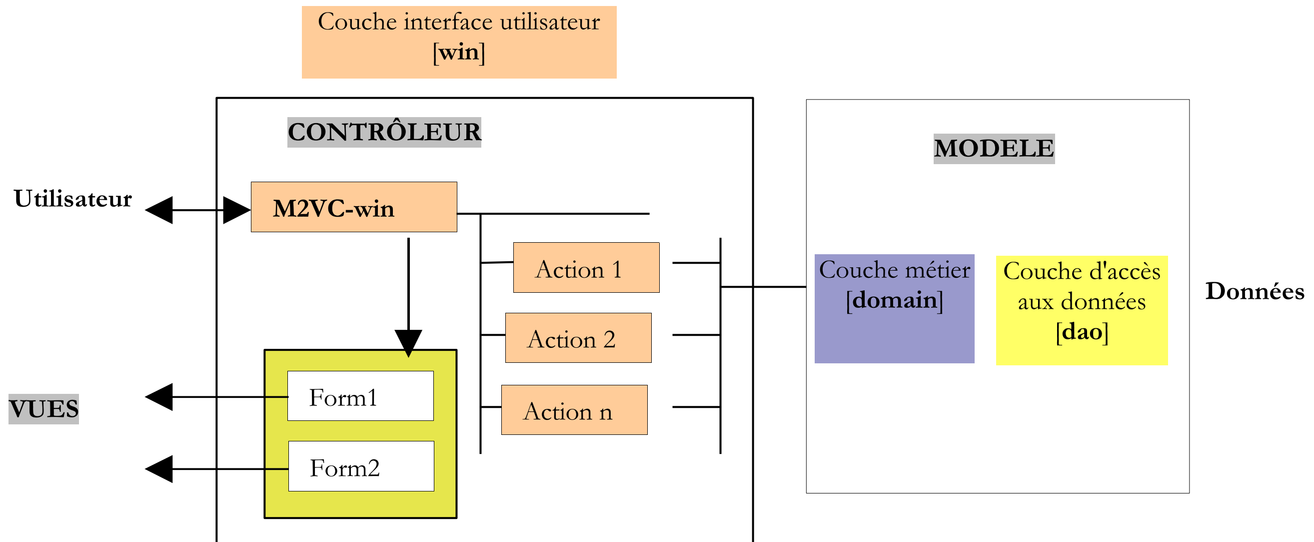
```
Namespace istia.st.cmv
  Public Class InfosAction
    ' champs privés
    Private _action As IAction
    Private _vue As IVue
    Private _états As Hashtable

    ' propriétés publiques associées
    Public Property action() As IAction
      Get
        Return _action
      End Get
      Set(ByVal Value As IAction)
        _action = Value
      End Set
    End Property

    Public Property vue() As IVue
      Get
        Return _vue
      End Get
      Set(ByVal Value As IVue)
        _vue = Value
      End Set
    End Property

    Public Property états() As Hashtable
      Get
        Return _états
      End Get
      Set(ByVal Value As Hashtable)
        _états = Value
      End Set
    End Property

  End Class

End Namespace
```

C'est un code simple. La classe [InfosAction] a simplement quatre propriétés publiques destinées au contrôleur :

`action`	une instance de classe implémentant [Iaction] à utiliser pour exécuter l'action. Peut être égal à [nothing]. Dans ce cas, l'attribut [vue] ci-dessous doit être initialisé.
`vue`	une instance de classe implémentant [Ivue]. Désigne une vue à afficher si l'attribut [action] ci-dessus n'a pas été initialisé.
`états`	un dictionnaire d'états. N'est utile que si l'attribut [action] a été initialisé. Dans ce cas, une méthode [Iaction.execute] a été lancée. Celle-ci rend une chaîne de caractères représentant le résultat de l'action. Avec ce résultat, le contrôleur va symboliser le nouvel état de l'application en présentant à l'utilisateur la vue associée à cet état. Le dictionnaire [états] associe donc une vue [Ivue] à un résultat d'action de type [String]

Les instances de [InfosAction] seront créées par configuration avec [Spring IoC].

Pour comprendre le rôle de [InfosAction], revenons sur l'architecture de [M2VC-win] :



1. le contrôleur [M2VC-win] reçoit une demande de l'utilisateur. Celle-ci viendra sous la forme d'une chaîne de caractères indiquant l'action à entreprendre.
2. [M2VC-win] récupère l'instance [InfosAction] liée à cette action. Pour cela, il aura un dictionnaire associant le nom d'une action à une instance [InfosAction] rassemblant les informations nécessaires à cette action.
3. si l'attribut [vue] de [InfosAction] est non vide, alors la vue associée est affichée par [vue.affiche]. On passe ensuite à l'étape 7.
4. si l'attribut [action] de [InfosAction] est non vide, alors l'action est exécutée par [action.execute].
5. la méthode [action.execute] fait ce qu'elle a à faire puis rend au contrôleur une chaîne de caractères représentant le résultat auquel elle est parvenue.
6. le contrôleur utilise le dictionnaire [états] de [InfosAction] pour trouver la vue V à afficher. Il l'affiche par [V.affiche]
7. ici une vue a été affichée. Le contrôleur s'est synchronisé avec et attend que la vue déclenche une nouvelle action. Celle-ci va être déclenchée par une action particulière de l'utilisateur sur la vue, qui à cette occasion va repasser la main au contrôleur en lui donnant le nom de l'action à exécuter.
8. le contrôleur reprend à l'étape 1

4.6 La classe [BaseControleur]

Il ne nous reste plus qu'à présenter le code du contrôleur de base pour comprendre comment tout ce puzzle fonctionne :

```
1. Imports System.Threading
2.
3. ' contrôleur de l'application
4. Namespace istia.st.cmv
5.  Public Class BaseControleur
6.    Implements IControleur
7.
8.    ' les propriétés du contrôleur
9.
10.   ' les actions à contrôler
11.   Private _actions As Hashtable
12.   Public Property actions() As Hashtable
13.     Get
14.       Return _actions
15.     End Get
16.     Set(ByVal Value As Hashtable)
17.       _actions = Value
18.     End Set
19.   End Property
20.
21.   ' le nom de la première action
22.   Private _firstActionName As String
23.   Public Property firstActionName() As String
24.     Get
25.       Return _firstActionName
26.     End Get
27.     Set(ByVal Value As String)
28.       _firstActionName = Value
29.     End Set
30.   End Property
31.
32.   ' le nom de la dernière action
33.   Private _lastActionName As String
34.   Public Property lastActionName() As String
```

```
35.    Get
36.      Return _lastActionName
37.    End Get
38.    Set(ByVal Value As String)
39.      _lastActionName = Value
40.    End Set
41.  End Property
42.
43.  ' la synchro avec les vues
44.  Private _synchro As AutoResetEvent
45.  Public Property synchro() As AutoResetEvent
46.    Get
47.      Return _synchro
48.    End Get
49.    Set(ByVal Value As AutoResetEvent)
50.      _synchro = Value
51.    End Set
52.  End Property
53.
54.  ' le moteur d'exécution des actions
55.  Public Sub run() Implements IControleur.run
56.    ' variables locales
57.    Dim configAction As InfosAction
58.    Dim actionName As String = firstActionName
59.    Dim état As String
60.    Dim vue As IVue
61.    Dim vuePrécédente As IVue
62.
63.    ' boucle d'exécution des actions
64.    Do While actionName <> lastActionName
65.      ' on récupère la config de l'action
66.      If actions(actionName) Is Nothing Then
67.        Throw New Exception(String.Format("L'action [{0}] n'a pas été configurée...", actionName))
68.      Else
69.        configAction = CType(actions(actionName), InfosAction)
70.      End If
71.      ' exécution de l'action s'il y en a une
72.      If Not configAction.action Is Nothing Then
73.        ' exécution de l'action
74.        état = configAction.action.execute()
75.        ' on récupère la vue associée à l'état
76.        If configAction.états(état) Is Nothing Then
77.          Throw New Exception(String.Format("L'état [{0}] de l'action [{1}] n'a pas été configuré...",
   état, actionName))
78.        Else
79.          vue = CType(configAction.états(état), IVue)
80.        End If
81.      Else
82.        ' pas d'action - directement la vue
83.        état = ""
84.        vue = configAction.vue
85.      End If
86.      ' on cache la vue précédente si elle est différente de celle qui va venir
87.      If Not vue Is vuePrécédente AndAlso Not vuePrécédente Is Nothing Then
88.        vuePrécédente.cache()
89.      End If
90.      ' on initialise la vue à afficher
91.      initVue(actionName, état, vue.nom)
92.      ' on affiche la vue en se synchronisant sur elle
93.      synchro.Reset()
94.      vue.affiche()
95.      synchro.WaitOne()
96.      ' action suivante
97.      actionName = vue.action
98.      vuePrécédente = vue
99.    Loop
100.     ' on cache la dernière vue affichée
101.     If Not vue Is Nothing Then vue.cache()
102.     ' c'est fini
103.   End Sub
104.
105.   ' préparation d'une vue
106.   Protected Overridable Sub initVue(ByVal actionName As String, ByVal état As String, ByVal nomVue As
   String)
107.     ' à faire dans les classes dérivées
108.   End Sub
109.
110.  End Class
111.
112.End Namespace
```

Le contrôleur sera intancié par configuration à l'aide de [Spring Ioc]. Commençons par présenter les propriétés publiques du contrôleur.

`actions` lignes 11-19

un dictionnaire faisant le lien entre le nom d'une action et l'instance [InfosAction] à utiliser pour cette action. Initialisé par fichier de configuration.

`firstActionName` lignes 21-30

le nom de la première action à exécuter. En effet, les noms des actions sont normalement donnés par les vues. Or au départ, il n'y a pas de vue. Initialisé par fichier de configuration.

`lastActionName` lignes 32-41

le nom de la dernière action à exécuter. Lorsqu'une vue envoie cette action au contrôleur, celui-ci s'arrête. Initialisé par fichier de configuration.

`synchro` lignes 43-52

l'instance [AutoResetEvent] qui permet au contrôleur de se synchroniser avec les vues. Initialisé par fichier de configuration.

Le contrôleur implémente l'interface [IControleur] (ligne 6). Il a donc une méthode [run] (ligne 55). C'est cette méthode qui lance le contrôleur. Celui-ci, après quelques initialisation, passe son temps dans une boucle où il va exécuter les actions demandés par les vues. Il va commencer par la première, qui aura été positionnée par configuration (ligne 58). Il terminera par celle ayant été déclarée comme la dernière (ligne 64).

Inspectons la boucle d'exécution des actions :

```
1.      ' boucle d'exécution des actions
2.      Do While actionName <> lastActionName
3.        ' on récupère la config de l'action
4.        If actions(actionName) Is Nothing Then
5.          Throw New Exception(String.Format("L'action [{0}] n'a pas été configurée...", actionName))
6.        Else
7.          configAction = CType(actions(actionName), InfosAction)
8.        End If
9.        ' exécution de l'action s'il y en a une
10.       If Not configAction.action Is Nothing Then
11.         ' exécution de l'action
12.         état = configAction.action.execute()
13.         ' on récupère la vue associée à l'état
14.         If configAction.états(état) Is Nothing Then
15.           Throw New Exception(String.Format("L'état [{0}] de l'action [{1}] n'a pas été configuré...",
   état, actionName))
16.         Else
17.           vue = CType(configAction.états(état), IVue)
18.         End If
19.       Else
20.         ' pas d'action - directement la vue
21.         état = ""
22.         vue = configAction.vue
23.       End If
24.       ' on cache la vue précédente si elle est différente de celle qui va venir
25.       If Not vue Is vuePrécédente AndAlso Not vuePrécédente Is Nothing Then
26.         vuePrécédente.cache()
27.       End If
28.       ' on initialise la vue à afficher
29.       initVue(actionName, état, vue.nom)
30.       ' on affiche la vue en se synchronisant sur elle
31.       synchro.Reset()
32.       vue.affiche()
33.       synchro.WaitOne()
34.       ' action suivante
35.       actionName = vue.action
36.       vuePrécédente = vue
37.     Loop
38.     ' on cache la dernière vue affichée
39.     If Not vue Is Nothing Then vue.cache()
```

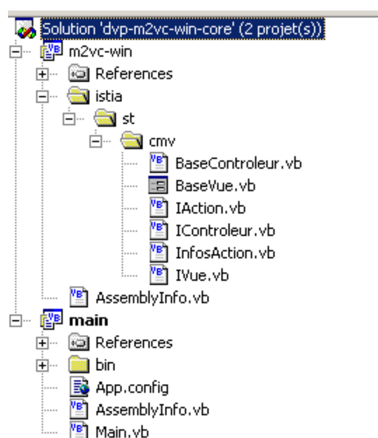
- l'attribut public [actions] du contrôleur contient les associations nom d'action <-> instance [InfosAction]. Le contrôleur commence donc par chercher dans son dictionnaire [actions] les informations concernant l'action à exécuter (lignes 4-8). S'il ne trouve rien, il lance une exception en expliquant l'erreur (ligne 5).
- une action peut soit demander l'exécution d'un objet [IAction] soit l'affichage d'une vue [Ivue].
- commençons par le cas où une instance [IAction] doit être exécutée. Dans ce cas, l'attribut [action] de l'instance [InfosAction] de l'action en cours n'est pas vide (ligne 10).
- l'instance [IAction] est alors exécutée (ligne 12). Cette exécution rend un résultat sous forme de chaîne de caractères placée ici dans la variable [état].
- on se rappelle que dans [InfosAction] on a un dictionnaire associant un état à une vue. Le contrôleur utilise ce dictionnaire pour récupérer l'instance [IVue] à afficher (lignes 14-18). Si cette instance n'est pas trouvée, une exception est lancée avec un message expliquant l'erreur (ligne 15).
- dans le cas où l'action ne demande que l'affichage d'une vue, celle-ci est récupérée lignes 19-22
- arrivé en ligne 24, on est prêt à afficher une vue [Ivue]. Si la vue à afficher n'est pas la même que la vue précédemment affichée, on cache cette dernière (lignes 24-27).

- on s'apprête à afficher une vue [IVue]. On veut laisser une chance au développeur de préparer celle-ci. On fait alors appel à une méthode [initVue] (ligne 29) en lui transmettant les informations dont on dispose :
  - le nom de l'action en cours
  - l'état rendu par cette action
  - le nom de la vue à afficher

  La méthode [initVue] a une implémentation locale qui ne fait rien (lignes 105-108 du code de la classe complète). On l'a déclarée redéfinissable (Overrides) afin que le développeur puisse la redéfinir s'il le souhaite.
- maintenant on peut afficher la vue. C'est fait dans les lignes 31-33. Le contrôleur positionne le verrou [synchro] à faux (ligne 31). Il affiche la vue (ligne 32). Il attend que le verrou [synchro] passe à vrai (ligne 33). C'est la vue affichée qui passera ce verrou à vrai lorsqu'elle voudra faire exécuter une nouvelle action au contrôleur. Nous avons déjà vu ce point.
- lorsque le contrôleur est débloqué (ligne 33), il récupère la nouvelle action à exécuter auprès de la vue qu'il a affichée (ligne 35).
- il note que cette vue devient désormais la vue précédente (ligne 36) avant de reboucler pour exécuter la nouvelle action (ligne 37).
- tout cela va durer un certain temps jusqu'à l'action à exécuter soit la dernière action [lastActionName] (ligne 2). On se retouve alors ligne 39 où on cache la dernière vue qu'on a affichée.
- le contrôleur a fini son travail. Il rend la main au programme qui l'a lancé (ligne 103 du code complet).

4.7 Le lanceur [m2vc-win.exe]

Rappelons la structure de la solution Visual Studio utilisée pour construire [M2VC-win] :



Le projet [main] de type [Console] donne naissance à l'exécutable [m2vc-win.exe]. C'est le lanceur du contrôleur. Il instancie tous les objets nécessaires à celui-ci à l'aide du fichier de configuration [m2vc-win.exe.config]. Cette instanciation est faite à l'aide de [Spring IoC]. Ci-dessus, le fichier de configuration s'appelle [App.config]. Lors de la génération du projet, VS le recopie dans le dossier [bin] du projet avec le nom [exécutable.config], donc ici [m2vc-win.exe.config].

Le code du lanceur [Main.vb] est le suivant :

```
1. Imports Spring.Context
2. Imports System.Configuration
3. Imports istia.st.cmv
4.
5. Module Main
6.  Sub main()
7.    ' variables locales
8.    Dim monControleur As IControleur
9.
10.   ' msg de patience
11.   Console.WriteLine("Application en cours d'initialisation. Patientez...")
12.   Try
13.     ' on active le controleur de l'application
14.     Dim contexte As IApplicationContext = CType(ConfigurationSettings.GetConfig("spring/context"),
   IApplicationContext)
15.     monControleur = CType(contexte.GetObject("controleur"), IControleur)
16.   Catch ex As Exception
17.     ' on s'arrête
18.     abort("Erreur lors de l'initialisation du contrôleur M2VC-win", ex, 1)
19.   End Try
20.   ' exécution application
21.   Console.WriteLine("Application lancée...")
22.   Try
23.     monControleur.run()
24.   Catch ex As Exception
```

```
25.     ' on affiche l'erreur
26.     abort("Erreur d'exécution", ex, 2)
27.   End Try
28.   ' fin normale
29.   Environment.Exit(0)
30. End Sub
31.
32. ' fin anormale
33. Sub abort(ByVal message As String, ByVal ex As Exception, ByVal exitCode As Integer)
34.   ' on affiche l'erreur
35.   Console.WriteLine(message)
36.   Console.WriteLine("---------------------------------------------------")
37.   Console.WriteLine(ex.ToString)
38.   Console.WriteLine("---------------------------------------------------")
39.   ' on laisse l'utilisateur voir le message
40.   Console.WriteLine("Tapez [enter] pour terminer...")
41.   Console.ReadLine()
42.   ' on s'arrête
43.   Environment.Exit(exitCode)
44.   Exit Sub
45. End Sub
46.End Module
```

- le contrôleur est créé lignes 12 à 19. On demande à [Spring] une instance d'un objet appelé [controleur] (ligne 15). Spring va alors exploiter le fichier [m2vc-win.exe.config] pour instancier le contrôleur. Nous alons voir bientôt la structure de ce fichier. Il est assez complexe à construire. Les erreurs de configuration vont générer des exceptions chez Spring. Le lanceur les affiche alors et s'arrête (lignes 16-18).
- si le contrôleur est instancié, il est lancé (ligne 23). A partir de maintenant, c'est le contrôleur qui a la main. Il ne la rendra qu'à deux occasions :
  - des erreurs de programmation ou de configuration qui engendrent des exceptions. Celles-ci sont affichées sur la console (lignes 24-27) et le programme est arrêté.
  - l'exécution de la dernière action [lastActionName]. Le lanceur s'arrête alors (ligne 29).

4.8 Le fichier de configuration [m2vc-win.exe.config]

[Spring IoC] nous permet d'avoir un contrôleur très simple. Cette simplicité est obtenue grâce au fichier de configuration [m2vc-win.exe.config]. Celui-ci n'est pas, au début, très simple à construire. Nous allons donner par la suite divers exemples. Examinons l'un d'entre-eux :

```
1. <?xml version="1.0" encoding="iso-8859-1" ?>
2. <configuration>
3.  <configSections>
4.    <sectionGroup name="spring">
5.      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
6.      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
7.    </sectionGroup>
8.  </configSections>
9.  <spring>
10.   <context type="Spring.Context.Support.XmlApplicationContext, Spring.Core">
11.     <resource uri="config://spring/objects" />
12.   </context>
13.   <objects>
14.     <!-- la session -->
15.     <object id="session" type="istia.st.cmv.appli.Session, appli" />
16.     <!-- la synchro -->
17.     <object id="synchro" type="System.Threading.AutoResetEvent,Mscorlib">
18.       <constructor-arg index="0">
19.         <value>true</value>
20.       </constructor-arg>
21.     </object>
22.     <!-- les vues -->
23.     <object id="vueSaisir" type="istia.st.cmv.appli.VueSaisir, appli">
24.       <property name="nom">
25.         <value>saisir</value>
26.       </property>
27.       <property name="session">
28.         <ref object="session" />
29.       </property>
30.       <property name="synchro">
31.         <ref object="synchro" />
32.       </property>
33.     </object>
34.     <object id="vueConsulter" type="istia.st.cmv.appli.VueConsulter, appli">
35.       <property name="nom">
36.         <value>consulter</value>
37.       </property>
38.       <property name="session">
39.         <ref object="session" />
40.       </property>
```

```
41.       <property name="synchro">
42.         <ref object="synchro" />
43.       </property>
44.     </object>
45.     <!-- la configuration des actions -->
46.     <object id="infosActionConsulter" type="istia.st.cmv.InfosAction, m2vc-win">
47.       <property name="vue">
48.         <ref object="vueConsulter" />
49.       </property>
50.     </object>
51.     <object id="infosActionSaisir" type="istia.st.cmv.InfosAction, m2vc-win">
52.       <property name="vue">
53.         <ref object="vueSaisir" />
54.       </property>
55.     </object>
56.     <!-- le contrôleur -->
57.     <object id="controleur" type="istia.st.cmv.BaseControleur, m2vc-win">
58.       <property name="synchro">
59.         <ref object="synchro" />
60.       </property>
61.       <property name="firstActionName">
62.         <value>saisir</value>
63.       </property>
64.       <property name="lastActionName">
65.         <value>quitter</value>
66.       </property>
67.       <property name="actions">
68.         <dictionary>
69.           <entry key="saisir">
70.             <ref object="infosActionSaisir" />
71.           </entry>
72.           <entry key="consulter">
73.             <ref object="infosActionConsulter" />
74.           </entry>
75.         </dictionary>
76.       </property>
77.     </object>
78.   </objects>
79. </spring>
80.</configuration>
```

- les lignes 1 à 12 sont standard. On les gardera toujours en l'état.
- la description du contrôleur commence ligne 13 où on commence à décrire tous les objets qui le composent.
- ligne 15 - on déclare un objet [Session]. Nous n'avons encore jamais rencontré cet objet. Ce n'est pas un objet propre au contrôleur mais à l'application particulière contrôlée ici. Nous retrouverons cet objet dans la plupart de nos exemples. En effet, le contrôleur fait interagir des vues [Ivue] et des actions [IAction] sans qu'on sache comment ces objets échangent de l'information. Il faut bien pourtant qu'elles en échangent puisqu'en général une vue affiche des informations produites par une action. C'est l'objet [Session] qui nous servira à cela. Il sera injecté dans chaque vue [Ivue] et chaque action [IAction]. Tous ces objets se partageront un même et unique objet [Session] qui leur servira à communiquer.
- lignes 17-21 - on déclare l'objet [synchro] de type [AutoResetEvent] qui sert à synchroniser le contrôleur et les vues. Cet objet sera toujours présent. Il doit être injecté dans chaque vue [Ivue] et dans le controleur [IControleur].
- lignes 22-44 - on déclare les vues [IVue] de l'application.
- lignes 23-33 - on déclare une vue identifiée par [id="vueSaisir"]. On sait qu'une vue a deux propriétés publiques : [nom] et [synchro]. Celles-ci sont initialisées lignes 24-26 et 30-32. Ces initialisations seront toujours à faire. La propriété [session] définie lignes 27-29 est propre à l'application particulière contrôlée ici et non pas une propriété initiale de [IVue]. Nous avons déjà expliqué son rôle.
- lignes 45-55 - on déclare les objets [InfosAction] des actions que sera amené à exécuter le contrôleur.
- lignes 46-50 - on déclare un objet [InfosAction] identifié par [id=infosActionConsulter]. On sait qu'un tel objet a trois propriétés publiques : **action** de type [IAction], **vue** de type [IVue], **états** de type [Hashtable]. L'objet [InfosAction] ici ne déclare qu'une vue (lignes 47-49). Cela signifie que sur cette action, le contrôleur affichera une vue. Il n'aura pas d'action [Iaction] à exécuter.
- lignes 51-55 - on déclare un objet [InfosAction] identifié par [id=infosActionConsulter] là aussi sans objet [IAction].
- lignes 57-77, on définit le contrôleur. Nous savons que le contrôleur a les propriétés publiques suivantes : le nom de la première action à exécuter [**firstActionName**] (lignes 61-63), le nom de la derrnière action à exécuter [**lastActionName**] (lignes 64-66), l'outil de synchronisation avec les vues [**synchro**] (lignes 58-60), le dictionnaire [**actions**] (lignes 67-77). Celui-ci est un dictionnaire associant un nom d'action (String) à un objet de type [InfosAction].
- dans cet exemple particulier, il n'y a aucune action [IAction] configurée.

Nous venons de décrire les grandes lignes du fichier de configuration [m2vc-win.exe.config]. La structure de celui-ci obéit aux règles de [Spring IoC]. Maîtriser celles-ci demande un peu de temps. Nous allons les revoir sur trois exemples.

5 Application 1

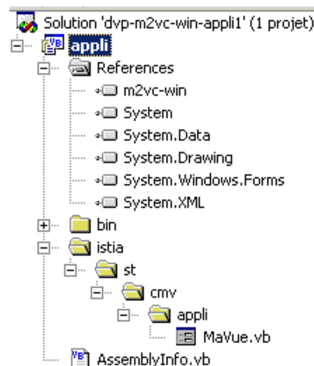
Notre première application n'affiche qu'une unique fenêtre :

C'est une fenêtre presque normale. Elle a cependant deux particularités :

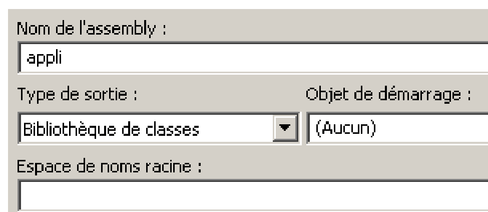
- elle ne se ferme pas
- le bouton [Quitter] fait appel au contrôleur. C'est lui qui va fermer la fenêtre

5.1 La structure

Un projet reposant sur le moteur [M2VC-win] doit produire une DLL. Sous VS, on a donc affaire à un projet de type [Bibliothèque de classes]. La solution VS de cette application a la structure suivante :

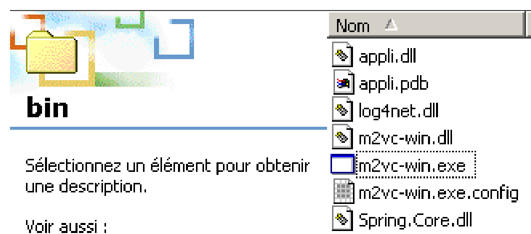


- le projet [appli] est défini comme suit :



La DLL produite s'appellera donc [appli.dll] et elle n'a pas d'espace de noms racine. Tous les objets du projet seront placés dans l'espace de noms [istia.st.cmv.appli]. Par souci d'homogénéité, ce sera ainsi pour tous les exemples à suivre.

- les fichiers nécessaires à l'exécution du contrôleur [m2vc-win] sont placés dans le dossier [bin] de l'application:



Les fichiers [m2vc-win.dll, m2vc-win.exe] viennent du projet [M2VC-win] étudié précédemment. Les fichiers [Spring.Core.dll, log4net.dll] viennent de Spring. Nous placerons systématiquement ces quatre fichiers dans le dossier [bin] de nos exemples.

- le projet [appli] référence (branche References du projet VS) le contrôleur [m2vc-win] qu'il trouve dans son dossier [bin]

5.2 Les vues

Comme nous l'avons dit, il n'y a qu'une vue :

Celle-ci est implémentée par la classe [MaVue.vb] suivante :

```
1. Imports istia.st.cmv
2. Imports System.Windows.Forms
3.
4. Namespace istia.st.cmv.appli
5.  Public Class MaVue
6.    Inherits BaseVue
7.
8. #Region " Code généré par le Concepteur Windows Form "
9.
10.....
11.   Friend WithEvents NumericUpDown1 As System.Windows.Forms.NumericUpDown
12.   Friend WithEvents ButtonQuitter As System.Windows.Forms.Button
13.....
14.
15.#End Region
16.
17.   ' affiche
18.   Public Overrides Sub affiche()
19.     ' on met le curseur flèche
20.     Me.Cursor = Cursors.Arrow
21.     ' position centrée sur écran
22.     Me.StartPosition = FormStartPosition.CenterScreen
23.     Me.WindowState = FormWindowState.Normal
24.     ' on dégèle le formulaire
25.     Me.Enabled = True
26.     ' on affiche la vue parent
27.     MyBase.affiche()
28.   End Sub
29.
30.     Private Sub ButtonQuitter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
   ButtonQuitter.Click
31.     ' action asynchrone - on gèle le formulaire
32.     Me.Enabled = False
33.     ' on passe l'action à la classe parent
34.     MyBase.exécute("quitter")
35.   End Sub
36. End Class
37.
38.End Namespace
```

- les lignes 11-12 définissent les deux composants du formulaire. Nous n'avons pas reproduit la région [#Région] qui est la région standard générée par VS.
- lignes 5-6 : la classe [MaVue] dérive de la classe [BaseVue] définie dans le contrôleur. C'est obligatoire.
- lignes 18-28 : la classe [MaVue] redéfinit la méthode [affiche]. Celle-ci fait des choses qui lui sont propres puis passe la main à sa classe parent (ligne 27). C'est obligatoire. Toute vue redéfinissant la méthode [affiche] pour ses propres besoins doit se terminer par l'appel à la méthode [affiche] de sa classe parent.
- un clic sur le bouton [Quitter] provoque l'exécution de la méthode [ButtonQuitter_Click], ligne 30. On s'apprête à exécuter l'action asynchrone "quitter". On décide de geler le formulaire (ligne 32) pour éviter que l'utilisateur ne provoque des événements pendant l'exécution de l'action asynchrone. Le formulaire sera réactivé lors d'un futur réaffichage (qui peut ne pas avoir lieu - c'est le cas ici) dans la méthode [affiche], ligne 25.

5.3 Le fichier de configuration

Le fichier de configuration [m2vc-win.exe.config] de l'application sera le suivant :

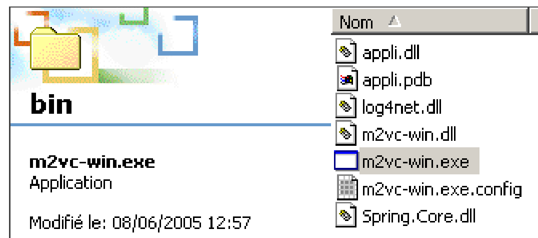
```
1. <?xml version="1.0" encoding="iso-8859-1" ?>
2. <configuration>
3.  <configSections>
4.    <sectionGroup name="spring">
5.      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
6.      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
7.    </sectionGroup>
8.  </configSections>
9.  <spring>
```

```
10.   <context type="Spring.Context.Support.XmlApplicationContext, Spring.Core">
11.     <resource uri="config://spring/objects" />
12.   </context>
13.   <objects>
14.     <!-- la synchro -->
15.     <object id="synchro" type="System.Threading.AutoResetEvent,Mscorlib">
16.       <constructor-arg index="0">
17.         <value>true</value>
18.       </constructor-arg>
19.     </object>
20.     <!-- les vues -->
21.     <object id="maVue" type="istia.st.cmv.appli.MaVue, appli">
22.       <property name="nom">
23.         <value>mavue</value>
24.       </property>
25.       <property name="synchro">
26.         <ref object="synchro" />
27.       </property>
28.     </object>
29.     <!-- les actions -->
30.     <!-- la configuration des actions -->
31.     <object id="infosActionMaVue" type="istia.st.cmv.InfosAction, m2vc-win">
32.       <property name="vue">
33.         <ref object="maVue" />
34.       </property>
35.     </object>
36.     <!-- le contrôleur -->
37.     <object id="controleur" type="istia.st.cmv.BaseControleur, m2vc-win">
38.       <property name="synchro">
39.         <ref object="synchro" />
40.       </property>
41.       <property name="firstActionName">
42.         <value>afficherMaVue</value>
43.       </property>
44.       <property name="lastActionName">
45.         <value>quitter</value>
46.       </property>
47.       <property name="actions">
48.         <dictionary>
49.           <entry key="afficherMaVue">
50.             <ref object="infosActionMaVue" />
51.           </entry>
52.         </dictionary>
53.       </property>
54.     </object>
55.   </objects>
56. </spring>
57.</configuration>
```

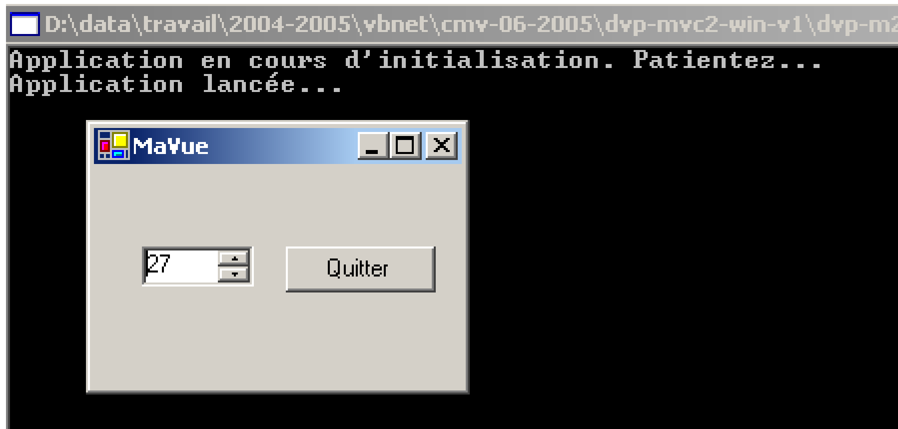
- lignes 15-19 : on définit l'objet [synchro] qui synchronise le contrôleur et les vues.
- lignes 21-28 : l'unique vue [maVue] est définie avec ses propriétés [nom] et [synchro]
- ligne 29 : pas d'objets [IAction]
- lignes 31-35 : les objets [InfosAction] sont définis. Il n'y en a qu'un. L'objet [infosActionMaVue] demande un affichage de la vue [maVue]
- lignes 37-54 : le contrôleur est défini.
- la propriété [synchro] est définie lignes 38-40. C'est bien sûr le même objet [synchro] que pour la vue [maVue]
- la propriété [firstActionName] est définie lignes 41-43. Il s'agit de l'action nommée " afficherMaVue "
- la propriété [lastActionName] est définie lignes 44-46. Il s'agit de l'action nommée "quitter"
- la liste des actions autres que [lastActionName] que le contrôleur doit gérer est définie lignes 47-53. C'est ainsi que l'action nommée " afficherMaVue " est associée à l'objet [infosActionMaVue]. Si on passe à la définition de cet objet (lignes 31-34), on voit que l'action nommée " afficherMaVue " va au final provoquer l'affichage de la vue [maVue], donc l'affichage de notre formulaire.
- lorsque l'utilisateur va cliquer sur le bouton [Quitter], nous avons vu que le formulaire va alors demander au contrôleur l'exécution de l'action appelée " quitter ". Comme celle-ci est définie comme la dernière action du contrôleur [lastActionName], celui-ci va terminer l'application.

5.4 Les tests

Une fois générée la solution, nous avons le dossier [bin] suivant :



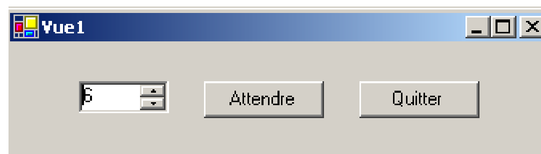
Un double-clic sur [m2vc-win.exe] lance l'application qui commence par afficher la fenêtre [MaVue] comme il a été expliqué :



La fenêtre [MaVue] a un comportement normal sauf qu'elle ne se ferme pas. Le bouton [Quitter] termine l'application.

6 Application 2

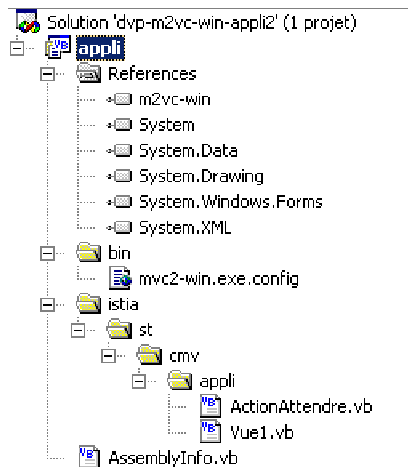
Notre deuxième application va, là encore, n'afficher qu'une fenêtre :



L'application est analogue à la précédente. Nous avons ajouté un bouton [Attendre] qui va déclencher une action qui sera exécutée par le contrôleur. Celui-ci fera alors exécuter une action [IAction] particulière. On va donc introduire la notion d'objets [IAction] qui ne l'avait pas encore été. L'objet [IAction] associé au bouton [Attendre] ne fera rien si ce n'est d'attendre 5 secondes. Avant de passer la main au contrôleur, lors du clic sur le bouton [Attendre], la vue [Vue1] va inhiber les boutons [Attendre] et [Quitter] mais pas l'incrémenteur. L'utilisateur va donc pouvoir pendant les 5 secondes d'attente jouer avec l'incrémenteur. On veut mettre en lumière ici, l'aspect asynchrone des actions exécutées par le contrôleur.

6.1 La structure

Le projet VS de l'application a la structure suivante :



6.2 Les vues

L'unique vue est implémentée par la classe [Vue1.vb] :

```
1. Imports istia.st.cmv
2. Imports System.Windows.Forms
3.
4. Namespace istia.st.cmv.appli
5.  Public Class Vue1
6.    Inherits BaseVue
7.
8. #Region " Code généré par le Concepteur Windows Form "
9.
10.....
11.   Friend WithEvents ButtonAttente As System.Windows.Forms.Button
12.   Friend WithEvents NumericUpDown1 As System.Windows.Forms.NumericUpDown
13.   Friend WithEvents ButtonQuitter As System.Windows.Forms.Button
14.....
15.#End Region
16.
17.   ' affiche
18.   Public Overrides Sub affiche()
19.     ' on met le curseur flèche
20.     Me.Cursor = Cursors.Arrow
21.     ' position centrée sur écran
22.     Me.StartPosition = FormStartPosition.CenterScreen
23.     Me.WindowState = FormWindowState.Normal
24.     ' on autorise le formulaire
25.     ButtonQuitter.Enabled = True
26.     ButtonAttente.Enabled = True
27.     ' on affiche la vue parent
28.     MyBase.affiche()
29.   End Sub
30.
31.   Private Sub ButtonAttente_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
   ButtonAttente.Click
32.     ' action asynchrone - on gèle le formulaire
33.     freeze()
34.     ' on passe la main au contrôleur
35.     MyBase.exécute("attendre")
36.   End Sub
37.
38.   Private Sub ButtonQuitter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
   ButtonQuitter.Click
39.     ' on passe la main au contrôleur
40.     MyBase.exécute("quitter")
41.   End Sub
42.
43.   ' gel du formulaire
44.   Private Sub freeze()
45.     ' on inhibe certains composants
46.     ButtonQuitter.Enabled = False
47.     ButtonAttente.Enabled = False
48.     ' on met le sablier
49.     Me.Cursor = Cursors.WaitCursor
50.   End Sub
51.
52. End Class
53.
54.End Namespace
```

- lignes 18-29, on définit la méthode [affiche] de la vue. Après quelques opérations qui lui sont propres, elle affiche sa classe de base (ligne 28). On rappelle que c'est obligatoire.
- lignes 38-41, on définit la gestion du clic sur le bouton [Quitter]. On demande au contrôleur d'exécuter l'action " quitter ". Cette action sera définie comme la dernière [lastActionName] du contrôleur et l'application se terminera donc.
- lignes 31-36, on définit la gestion du clic sur le bouton [Attendre]. On demande au contrôleur d'exécuter l'action "attendre". Avant de passer la main au contrôleur, les boutons [Attendre] et [Quitter] sont gelés et le sablier affiché (lignes 44-50). Les boutons retrouveront un état actif, lorsque la vue sera réaffichée (lignes 25-26). L'incrémenteur n'ayant pas été gelé, on devrait pouvoir l'utiliser pendant que le contrôleur exécute l'action " attendre ".

6.3 Les actions

L'unique action est implémentée par [ActionAttendre.vb] :

```
1. Imports istia.st.cmv
2. Imports System.Threading
3.
4. Namespace istia.st.cmv.appli
5.  Public Class ActionAttendre
6.    Implements IAction
7.
8.    Public Function execute() As String Implements IAction.execute
9.      ' implémente l'action [attente] - on attend 5 secondes
10.     Thread.Sleep(5000)
11.     ' on retourne le résultat
12.     Return "succès"
13.   End Function
14. End Class
15.
16.End Namespace
```

- ligne 6 : la classe [ActionAttente] implémente l'interface [IAction]. C'est obligatoire.
- elle doit donc définir la méthode [execute] qui sera appelée par le contrôleur. C'est ce qui est fait lignes 8-13.
- dans [execute], on ne fait rien sinon attendre 5 secondes (ligne 10). La méthode [execute] doit rendre une chaîne de caractères comme résultat. Souvent nous utiliserons les résultats " succès " et " échec " pour symboliser la réussite ou non de l'action. Parfois le résultat d'une action peut être plus fin que succès/échec. On aura alors plusieurs résultats possibles. On choisira pour chacun d'eux un nom significatif. Ici rien ne peut échouer. On rend la chaîne " succès ".

6.4 Le fichier de configuration

Le fichier de configuration [m2vc-win.exe.config] de l'application est le suivant :

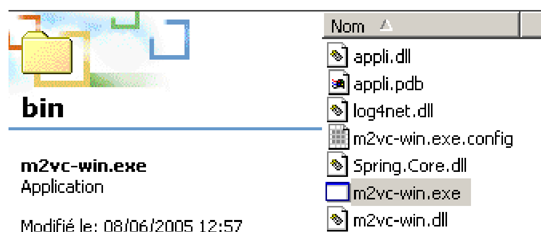
```
1.<?xml version="1.0" encoding="iso-8859-1" ?>
2. <configuration>
3.  <configSections>
4.    <sectionGroup name="spring">
5.      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
6.      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
7.    </sectionGroup>
8.  </configSections>
9.  <spring>
10.   <context type="Spring.Context.Support.XmlApplicationContext, Spring.Core">
11.     <resource uri="config://spring/objects" />
12.   </context>
13.   <objects>
14.     <!-- la synchro -->
15.     <object id="synchro" type="System.Threading.AutoResetEvent,Mscorlib">
16.       <constructor-arg index="0">
17.         <value>true</value>
18.       </constructor-arg>
19.     </object>
20.     <!-- les vues -->
21.     <object id="vue1" type="istia.st.cmv.appli.Vue1, appli">
22.       <property name="nom">
23.         <value>vue1</value>
24.       </property>
25.       <property name="synchro">
26.         <ref object="synchro" />
27.       </property>
28.     </object>
29.     <!-- les actions -->
30.     <object id="actionAttendre" type="istia.st.cmv.appli.ActionAttendre, appli" />
31.     <!-- la configuration des actions -->
```

```
32.     <object id="infosActionCommencer" type="istia.st.cmv.InfosAction, m2vc-win">
33.       <property name="vue">
34.         <ref object="vue1" />
35.       </property>
36.     </object>
37.     <object id="infosActionAttendre" type="istia.st.cmv.InfosAction, m2vc-win">
38.       <property name="action">
39.         <ref object="actionAttendre" />
40.       </property>
41.       <property name="états">
42.         <dictionary>
43.           <entry key="succès">
44.             <ref object="vue1" />
45.           </entry>
46.         </dictionary>
47.       </property>
48.     </object>
49.     <!-- le contrôleur -->
50.     <object id="controleur" type="istia.st.cmv.BaseControleur, m2vc-win">
51.       <property name="synchro">
52.         <ref object="synchro" />
53.       </property>
54.       <property name="firstActionName">
55.         <value>commencer</value>
56.       </property>
57.       <property name="lastActionName">
58.         <value>quitter</value>
59.       </property>
60.       <property name="actions">
61.         <dictionary>
62.           <entry key="commencer">
63.             <ref object="infosActionCommencer" />
64.           </entry>
65.           <entry key="attendre">
66.             <ref object="infosActionAttendre" />
67.           </entry>
68.         </dictionary>
69.       </property>
70.     </object>
71.   </objects>
72. </spring>
73.</configuration>
```

- l'objet [synchro] nécessaire à la synchronisation du contrôleur et des vues est défini lignes 15-19
- l'unique vue [vue1] est définie lignes 21-28
- l'unique action [actionAttendre] est définie ligne 30
- les informations sur les actions sont données lignes 32-48
- l'objet [infosActionCommencer] sera associé à la première action du contrôleur. Les lignes 32-36 disent que sur cette action, la vue [vue1] doit être affichée.
- l'objet [infosActionAttendre] sera associé à l'action " attendre ", celle qui sera déclenchée par le clic sur le bouton [Attendre]. Les lignes 37-48 disent les choses suivantes :
  - lignes 38-40 : que le contrôleur doit commencer par exécuter l'action [actionAttendre] définie ligne 30.
  - lignes 41-47 : que sur le résultat " succès " de l'action [actionAttendre], la vue [vue1] doit être affichée. On voit donc qu'après l'attente de 5 secondes, la vue [vue1] sera réaffichée.
- le contrôleur est défini lignes 50-70. Les attributs [synchro, firstActionName, lastActionName] sont définis lignes 51-59, la liste des actions lignes 60-69.

6.5 Les tests

Une fois générée la solution, nous avons le dossier [bin] suivant :



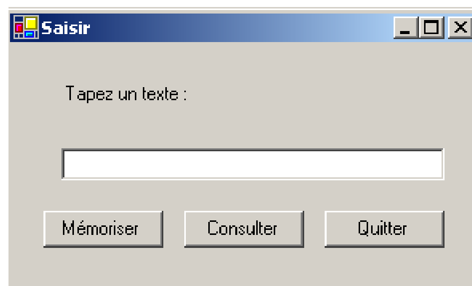
Un double-clic sur [m2vc-win.exe] lance l'application :

- l'incrémenteur s'utilise normalement
- le clic sur le bouton [Attendre] change la vue :

- pendant les 5 secondes de présence du sablier, on constate que l'incrémenteur est actif. Il suffit de cliquer sur sa barre de défilement avec le sablier pour s'en rendre compte.
- un clic sur le bouton [Quitter] de la vue redevenue active arrête l'application.

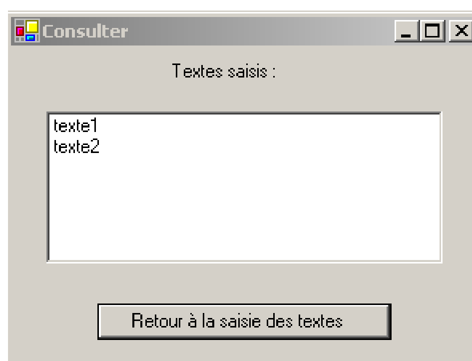
7 Application 3

Nous abordons ici une application à trois vues. L'application commence par afficher la vue [Saisir] suivante :



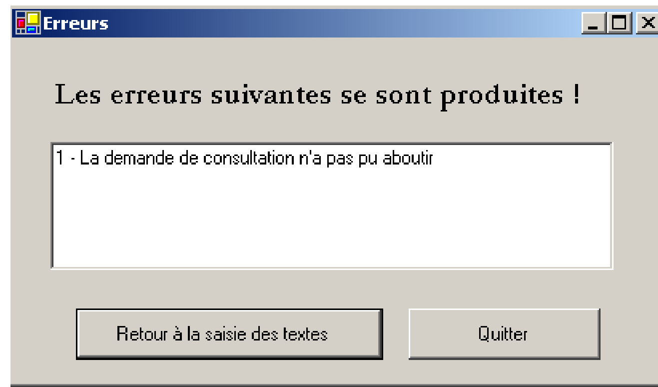
L'utilisateur entre des textes dans le champ de saisie. Le bouton [Mémoriser] les mémorise au fur et à mesure. Le bouton [Consulter] sert à voir la liste des textes saisis. L'action associée a été construite pour échouer en moyenne une fois sur deux.

Lorsque l'action [Consulter] réussit, on a la vue [Consulter] suivante :

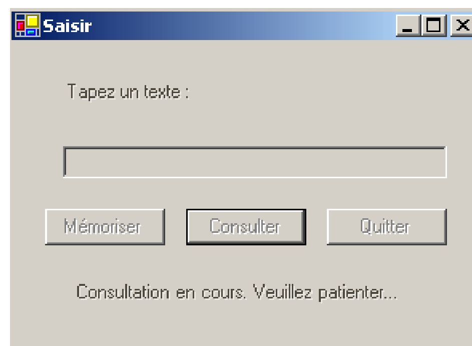


Un bouton permet de revenir à la saisie des textes.

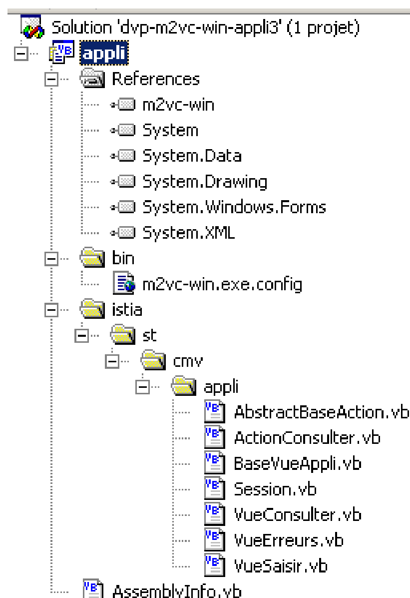
Lorsque l'action [Consulter] échoue, on a la vue [Erreurs] suivante :



On peut alors soit revenir à la saisie des textes, soit quitter l'application. En cas d'erreurs, l'action [Consulter] a été construite pour attendre trois secondes pour rendre son résultat. Pendant cette attente, la vue [Saisir] est complètement gelée. Elle affiche un message pour faire patienter l'utilisateur :



7.1 La structure de l'application



7.2 La session

Dans les exemples étudiés précédemment, les actions [IAction] et les vues [IVue] n'échangeaient pas d'informations. Ce n'est pas le cas courant. Dans cette application, la vue [VueSaisir] remplit un objet [ArrayList] avec les textes qu'elle a saisis. Lorsque la vue [VueConsulter] va vouloir afficher les textes saisis, elle devra avoir accès à cet objet [ArrayList]. Le contrôleur [M2VC-win] ne donne aucune aide pour ces échanges d'informations entre vues et actions. C'est au développeur d'organiser ceux-ci. Nous proposons ici une méthode simple qui devrait suffire pour beaucoup d'applications. Il s'agit de créer un unique objet qui contiendra toutes les informations à partager entre les vues et les actions. Il n'y a pas de problème de synchronisation. Les objets du contrôleur

ne sont jamais amenés à accéder à cet objet partagé de façon simultanée. Nous appellerons cet objet [Session] par similitude avec l'objet [Session] des applications web dans lequel on stocke tout ce qu'on veut garder au fil des échanges client-serveur. Tous les objets d'une application web ont accès à cet objet [Session] comme ce sera le cas ici.

Notre classe [Session] sera la suivante :

```
Namespace istia.st.cmv.appli

  Public Class Session
    Public textes As New ArrayList
    Public erreurs As New ArrayList
  End Class
End Namespace
```

Elle contiendra la liste des textes mémorisés par la vue [VueSaisir] et les erreurs à afficher par la vue [VueErreurs].

7.3 Les vues

Il ya trois vues. Elles sont implémentées par les fichiers [BaseVueAppli.vb, VueConsulter.vb, VueErreurs.vb, VueSaisir.vb].

7.3.1 La vue [BaseVueAppli]

La vue [BaseVueAppli] est une classe de base pour les vues [VueConsulter, VueErreurs, VueSaisir]. On y a mis le comportement commun des trois vues :

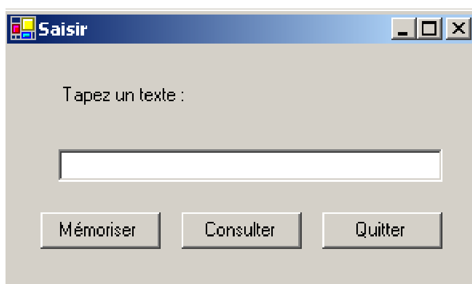
```
1. Imports istia.st.cmv
2. Imports System.Windows.Forms
3.
4. Namespace istia.st.cmv.appli
5.
6.  Public Class BaseVueAppli
7.    Inherits BaseVue
8.
9. #Region " Code généré par le Concepteur Windows Form "
10. ....
11.#End Region
12.
13.   ' session de l'application
14.   Private _session As Session
15.   Public Property session() As Session
16.     Get
17.       Return _session
18.     End Get
19.     Set(ByVal Value As Session)
20.       _session = Value
21.     End Set
22.   End Property
23.
24.   ' affiche
25.   Public Overrides Sub affiche()
26.     ' on met le curseur flèche
27.     Me.Cursor = Cursors.Arrow
28.     ' position centrée sur écran
29.     Me.StartPosition = FormStartPosition.CenterScreen
30.     Me.WindowState = FormWindowState.Normal
31.     ' on autorise le formulaire
32.     Me.Enabled = True
33.     ' on affiche la vue parent
34.     MyBase.affiche()
35.   End Sub
36.
37.   ' exécute
38.   Protected Sub exécuteAction(ByVal action As String)
39.     ' action asynchrone - on gèle le formulaire
40.     Me.Enabled = False
41.     ' on passe la main à la classe parent
42.     MyBase.exécute(action)
43.   End Sub
44.
45. End Class
46.End Namespace
```

- l'objet [Session] défini précédemment devra être injecté dans tous les actions [IAction] et les vues [IVue]. La classe [BaseVueAppli] définit donc une propriété publique appelée [session] pour intégrer cet objet (lignes 14-22).
- la méthode [affiche] (lignes 25-35) définit un mode d'affichage standard pour toutes les vues de l'application. Elle se termine par l'affichage de la classe de base (ligne 34). C'est obligatoire.

- la méthode [exécuteAction] a pour but de donner un comportement standard à l'exécution des actions asynchrones des différentes vues :
  - la vue est gelée - ligne 40
  - l'exécution de l'action est déléguée à la classe de base [BaseVue] (ligne 42).
  - la vue retrouvera un état actif lorsqu'elle sera réaffichée - ligne 32

7.3.2 La vue [VueSaisir]

Rappelons l'aspect visuel ce cette vue :



Le code de la classe [VueSaisir] est le suivant :

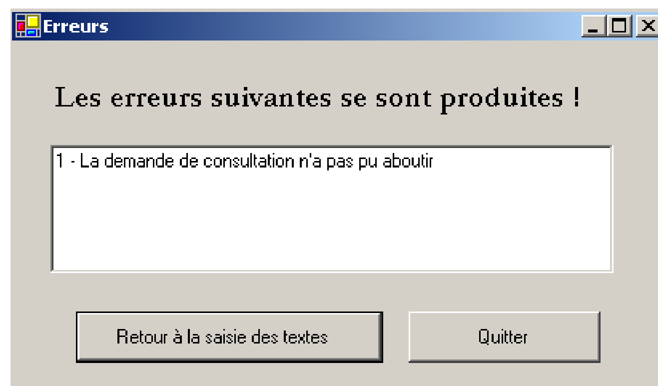
```
1. Imports istia.st.cmv
2. Imports System.Windows.Forms
3.
4. Namespace istia.st.cmv.appli
5.  Public Class VueSaisir
6.    Inherits BaseVueAppli
7.
8. #Region " Code généré par le Concepteur Windows Form "
9.
10....
11.   Friend WithEvents Label1 As System.Windows.Forms.Label
12.   Friend WithEvents TextBoxSaisie As System.Windows.Forms.TextBox
13.   Friend WithEvents ButtonMémoriser As System.Windows.Forms.Button
14.   Friend WithEvents ButtonConsulter As System.Windows.Forms.Button
15.   Friend WithEvents ButtonQuitter As System.Windows.Forms.Button
16.   Friend WithEvents LabelEtat As System.Windows.Forms.Label
17.....
18.#End Region
19.
20.   Public Overrides Sub affiche()
21.     ' on cache le label d'état
22.     LabelEtat.Visible = False
23.     ' affiche classe parente
24.     MyBase.affiche()
25.   End Sub
26.
27.   ' on mémorise le texte saisi
28.   Private Sub ButtonMémoriser_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
   Handles ButtonMémoriser.Click
29.     Dim texte As String = TextBoxSaisie.Text.Trim
30.     ' texte non vide
31.     If texte = "" Then
32.       MessageBox.Show("Vous n'avez pas tapé de texte", "Erreur", MessageBoxButtons.OK,
   MessageBoxIcon.Information)
33.       Exit Sub
34.     End If
35.     ' mémorisation dans la session
36.     session.textes.Add(texte)
37.     ' nettoyage
38.     TextBoxSaisie.Text = ""
39.   End Sub
40.
41.   Private Sub ButtonConsulter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
   Handles ButtonConsulter.Click
42.     ' on affiche le label d'état
43.     LabelEtat.Visible = True
44.     ' on passe la main au contrôleur
45.     MyBase.exécuteAction("consulter")
46.   End Sub
47.
48.   Private Sub ButtonQuitter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
   ButtonQuitter.Click
49.     ' on passe la main au contrôleur
50.     MyBase.exécuteAction("quitter")
51.   End Sub
```

```
52. End Class
53.
54.End Namespace
```

- lignes 5-6 : la classe [VueSaisir] dérive de la classe [BaseVueAppli]. Rappelons que cette dernière dérive elle-même de la classe [BaseVue] de [M2VC-win].
- lignes 20-25 : la méthode [affiche] est définie. Elle ne fait que ce qui est propre à la vue puisqu'elle passe ensuite la main à sa classe de base [BaseVueAppli] (ligne 24). Rappelons que celle-ci va geler le formulaire puis passer à son tour la main à sa classe de base [BaseVue]. L'exécution de la méthode [VueSaisir.affiche] se termine donc par celle de la méthode [BaseVue.affiche]. C'est obligatoire.
- lignes 28-39 : on traite le clic sur le bouton [Mémoriser]. Le texte saisi par l'utilisateur est mémorisé dans l'attribut [textes] de l'objet [Session] de la vue (ligne 36). Cet objet est une propriété de la classe de base [BaseVueAppli].
- lignes 41-46 : on traite le clic sur le bouton [Consulter]. On rend visible le texte "Consultation en cours. Patientez..." (ligne 43) et on passe la main à la classe de base [BaseVueAppli] pour faire exécuter l'action asynchrone "consulter" (ligne 45). Le texte "Consultation en cours. Patientez..." sera caché lorsque la vue redeviendra active (ligne 22).
- lignes 48-51 : on traite le clic sur le bouton [Quitter]. On passe la main à la classe de base [BaseVueAppli] pour faire exécuter l'action asynchrone "quitter" (ligne 50).

7.3.3 La vue [VueConsulter]

Rappelons l'aspect visuel ce cette vue :



Le code de la classe [VueConsulter] est le suivant :

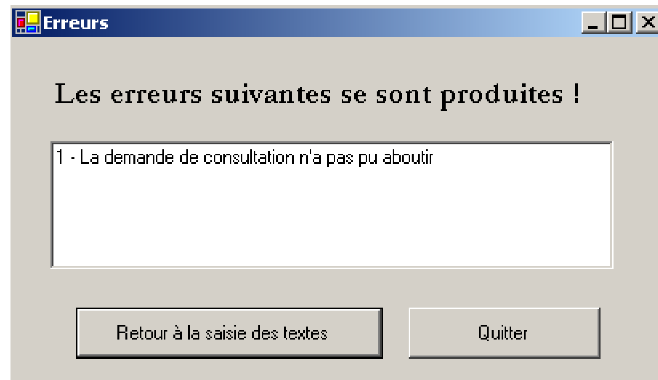
```
1. Imports istia.st.cmv
2.
3. Namespace istia.st.cmv.appli
4.  Public Class VueConsulter
5.    Inherits BaseVueAppli
6.
7. #Region " Code généré par le Concepteur Windows Form "
8.
9. ....
10.   Friend WithEvents Label1 As System.Windows.Forms.Label
11.   Friend WithEvents ListBoxTextes As System.Windows.Forms.ListBox
12.   Friend WithEvents ButtonRetour As System.Windows.Forms.Button
13.....
14.#End Region
15.
16.   Public Overrides Sub affiche()
17.     ' affiche les textes de la session
18.     ListBoxTextes.Items.Clear()
19.     For i As Integer = 0 To session.textes.Count - 1
20.       ListBoxTextes.Items.Add(session.textes(i))
21.     Next
22.     ' affiche classe parente
23.     MyBase.affiche()
24.   End Sub
25.
26.   Private Sub ButtonRetour_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
    ButtonRetour.Click
27.     ' on passe la main au contrôleur
28.     MyBase.exécuteAction("saisir")
29.   End Sub
30. End Class
31.End Namespace
```

- lignes 4-5 : la classe [VueConsulter] dérive de la classe [BaseVueAppli] qui dérive elle-même de la classe [BaseVue] de [M2VC-win].

* lignes 16-24 : la méthode [affiche] est définie. Elle fait ce qui est propre à la vue puis passe la main à sa classe de base [BaseVueAppli] (ligne 23). La méthode [affiche] locale affiche les textes mémorisés par la vue [VueSaisir]. Elle les trouve dans l'objet [Session] de sa classe de base, objet qu'elle partage avec toutes les autres vues.
* lignes 26-28 : on traite le clic sur le bouton [Retour...]. On passe la main à la classe de base [BaseVueAppli] pour faire exécuter l'action asynchrone "saisir" (ligne 28).

7.3.4 La vue [VueErreurs]

Rappelons l'aspect visuel ce cette vue :



Le code de la classe [VueErreurs] est le suivant :

```
1. Namespace istia.st.cmv.appli
2.  Public Class VueErreurs
3.    Inherits BaseVueAppli
4.
5. #Region " Code généré par le Concepteur Windows Form "
6.
7. ...
8.    Friend WithEvents Label1 As System.Windows.Forms.Label
9.    Friend WithEvents TextBoxErreurs As System.Windows.Forms.TextBox
10.   Friend WithEvents ButtonRetourSaisies As System.Windows.Forms.Button
11.   Friend WithEvents ButtonQuitter As System.Windows.Forms.Button
12....
13.#End Region
14.
15.   Private Sub ButtonRetourSaisies_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles ButtonRetourSaisies.Click
16.     ' retour à la saisie des textes
17.     MyBase.exécute("saisir")
18.   End Sub
19.
20.   Private Sub ButtonQuitter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
  ButtonQuitter.Click
21.     ' on quitte l'appli
22.     MyBase.exécuteAction("quitter")
23.   End Sub
24.
25.   ' affichage de la vue
26.   Public Overrides Sub affiche()
27.     ' affiche les erreurs
28.     Dim texte As String = ""
29.     For i As Integer = 0 To session.erreurs.Count - 1
30.       texte += String.Format("{0} - {1}{2}", (i + 1), session.erreurs(i).ToString, ControlChars.CrLf)
31.     Next
32.     ' mise en place du texte dans le textbox
33.     TextBoxErreurs.Text = texte
34.     ' affichage vue parent
35.     MyBase.affiche()
36.   End Sub
37. End Class
38.
39.End Namespace
```

* lignes 2-3 : la classe [VueErreurs] dérive de la classe [BaseVueAppli] qui dérive elle-même de la classe [BaseVue] de [M2VC-win].
* lignes 26-36 : la méthode [affiche] est définie. Elle fait ce qui est propre à la vue puis passe la main à sa classe de base [BaseVueAppli] (ligne 35). La méthode [affiche] locale affiche les erreurs construites par l'action [ActionConsulter] qui va être présentée prochainement. Elle trouve ces erreurs dans l'objet [Session] de sa classe de base, objet partagé entre toutes les vues et actions.

- lignes 15-18 : on traite le clic sur le bouton [Retour...]. On passe la main à la classe de base [BaseVueAppli] pour faire exécuter l'action asynchrone "saisir" (ligne 28).
- lignes 20-23 : on traite le clic sur le bouton [Quitter]. On passe la main à la classe de base [BaseVueAppli] pour faire exécuter l'action asynchrone "quitter" (ligne 22).

7.4 Les actions

Nous n'avons ici qu'une action. La classe [ActionConsulter] est chargée d'exécuter l'action asynchrone "consulter". Nous avons éprouvé le besoin de nous placer dans un cadre plus général où il y aurait plusieurs actions qui dériveraient toutes d'une classe abstraite [AbstractAction]. Celle-ci factoriserait tout ce que les différentes actions ont en commun.

7.4.1 L'action [AbstractAction]

Le code de [AbstractAction] est le suivant :

```
1. Imports istia.st.cmv
2.
3. Namespace istia.st.cmv.appli
4.  Public MustInherit Class AbstractBaseAction
5.    Implements IAction
6.
7.    ' la session commune aux actions et vues
8.    Private _session As Session
9.
10.   Public Property session() As Session
11.     Get
12.       Return _session
13.     End Get
14.     Set(ByVal Value As Session)
15.       _session = Value
16.     End Set
17.   End Property
18.
19.   ' fonction execute laissée à la charge des classes dérivées
20.   Public MustOverride Function execute() As String Implements IAction.execute
21. End Class
22.
23.End Namespace
```

- la classe implémente l'interface [IAction] (ligne 5). Elle doit donc implémenter la méthode [execute] de cette interface. C'est fait ligne 20. On ne sait pas quoi exécuter. Seules les classes dérivées le sauront. La méthode [execute] est donc marquée abstraite (MustOverride) ce qui entraîne que la classe est elle-même abstraite (attribut MustInherit, ligne 4). On rappelle qu'une classe abstaite est une classe qu'on doit obligatoirement dériver pour en avoir des instances.
- nous avons dit que la communication [actions-vues] se ferait via un unique objet [Session] partagé par tous les objets [actions-vues]. L'objet [Session] sera injecté dans [AbstractBaseAction] grâce à l'attribut public [session] (lignes 8-17).

7.4.2 L'action [ActionConsulter]

Le code de [ActionConsulter] est le suivant :

```
1. Imports istia.st.cmv
2. Imports System.Threading
3.
4. Namespace istia.st.cmv.appli
5.
6.  Public Class ActionConsulter
7.    Inherits AbstractBaseAction
8.
9.    ' générateur de nombres améatoires
10.   Private random As New [Random]
11.
12.   Public Overrides Function execute() As String
13.     ' on tire un nombre au hasard
14.     If random.Next(2) Mod 2 = 0 Then
15.       ' on patiente 3 secondes
16.       Thread.Sleep(3000)
17.       ' on signale une erreur
18.       With session.erreurs
19.         .Clear()
20.         .Add("La demande de consultation n'a pas pu aboutir")
21.       End With
22.       ' on rend l'état [erreurs]
23.       Return "erreurs"
24.     Else
25.        ' on rend l'état [consulter]
```

```
26.        Return "consulter"
27.    End If
28.   End Function
29. End Class
30.End Namespace
```

- lignes 6-7 : la classe [ActionConsulter] dérive de la classe [AbstractAction]
- lignes 12-28 : la classe [ActionConsulter] implémente la méthode [execute] que n'avait pas implémentée sa classe de base[AbstractAction]. Qu'y fait on ?
- ligne 14, on tire un nombre entier N au hasard dans [0,1].
- si N est pair (ligne 14), on s'arrête pendant 3 secondes (ligne 16). Puis lignes 18-21, on insère un message d'erreur arbitraire dans l'attribut [erreurs] de l'objet [Session] de la classe de base [AbstractAction].
- ligne 23 - l'action est terminée. On rend la chaîne "erreurs" pour signaler qu'on a rencontré des problèmes.
- si N est impair, on ne fait rien et on rend directement la chaîne "consulter" pour signaler que tout s'est bien passé (ligne 26).

Les lois du hasard disent que sur 100 actions "consulter", aux alentours de 50 d'entre-elles devraient revenir avec l'état "consulter" et aux alentours de 50 autres devraient revenir avec l'état "erreurs".

7.5 Le fichier de configuration

Le fichier de configuration [m2vc-win.exe.config] de l'application est le suivant :

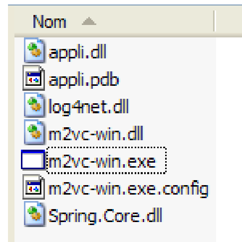
```
1. <?xml version="1.0" encoding="iso-8859-1" ?>
2. <configuration>
3.  <configSections>
4.    <sectionGroup name="spring">
5.      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
6.      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
7.    </sectionGroup>
8.  </configSections>
9.  <spring>
10.   <context type="Spring.Context.Support.XmlApplicationContext, Spring.Core">
11.     <resource uri="config://spring/objects" />
12.   </context>
13.   <objects>
14.     <!-- la synchro -->
15.     <object id="synchro" type="System.Threading.AutoResetEvent,Mscorlib">
16.       <constructor-arg index="0">
17.         <value>true</value>
18.       </constructor-arg>
19.     </object>
20.     <!-- la session -->
21.     <object id="session" type="istia.st.cmv.appli.Session, appli" />
22.     <!-- les vues -->
23.     <object id="vueSaisir" type="istia.st.cmv.appli.VueSaisir, appli">
24.       <property name="nom">
25.         <value>saisir</value>
26.       </property>
27.       <property name="session">
28.         <ref object="session" />
29.       </property>
30.       <property name="synchro">
31.         <ref object="synchro" />
32.       </property>
33.     </object>
34.     <object id="vueConsulter" type="istia.st.cmv.appli.VueConsulter, appli">
35.       <property name="nom">
36.         <value>consulter</value>
37.       </property>
38.       <property name="session">
39.         <ref object="session" />
40.       </property>
41.       <property name="synchro">
42.         <ref object="synchro" />
43.       </property>
44.     </object>
45.     <object id="vueErreurs" type="istia.st.cmv.appli.VueErreurs, appli">
46.       <property name="nom">
47.         <value>erreurs</value>
48.       </property>
49.       <property name="session">
50.         <ref object="session" />
51.       </property>
52.       <property name="synchro">
53.         <ref object="synchro" />
54.       </property>
55.     </object>
56.     <!-- les actions -->
57.     <object id="actionConsulter" type="istia.st.cmv.appli.ActionConsulter, appli">
58.       <property name="session">
```

```
59.         <ref object="session" />
60.       </property>
61.     </object>
62.     <!-- la configuration des actions -->
63.     <object id="infosActionConsulter" type="istia.st.cmv.InfosAction, m2vc-win">
64.       <property name="action">
65.         <ref object="actionConsulter" />
66.       </property>
67.       <property name="états">
68.         <dictionary>
69.           <entry key="erreurs">
70.             <ref object="vueErreurs" />
71.           </entry>
72.           <entry key="consulter">
73.             <ref object="vueConsulter" />
74.           </entry>
75.         </dictionary>
76.       </property>
77.     </object>
78.     <object id="infosActionSaisir" type="istia.st.cmv.InfosAction, m2vc-win">
79.       <property name="vue">
80.         <ref object="vueSaisir" />
81.       </property>
82.     </object>
83.     <!-- le contrôleur -->
84.     <object id="controleur" type="istia.st.cmv.BaseControleur, m2vc-win">
85.       <property name="synchro">
86.         <ref object="synchro" />
87.       </property>
88.       <property name="firstActionName">
89.         <value>saisir</value>
90.       </property>
91.       <property name="lastActionName">
92.         <value>quitter</value>
93.       </property>
94.       <property name="actions">
95.         <dictionary>
96.           <entry key="saisir">
97.             <ref object="infosActionSaisir" />
98.           </entry>
99.           <entry key="consulter">
100.             <ref object="infosActionConsulter" />
101.           </entry>
102.         </dictionary>
103.       </property>
104.     </object>
105.   </objects>
106.  </spring>
107.</configuration>
```

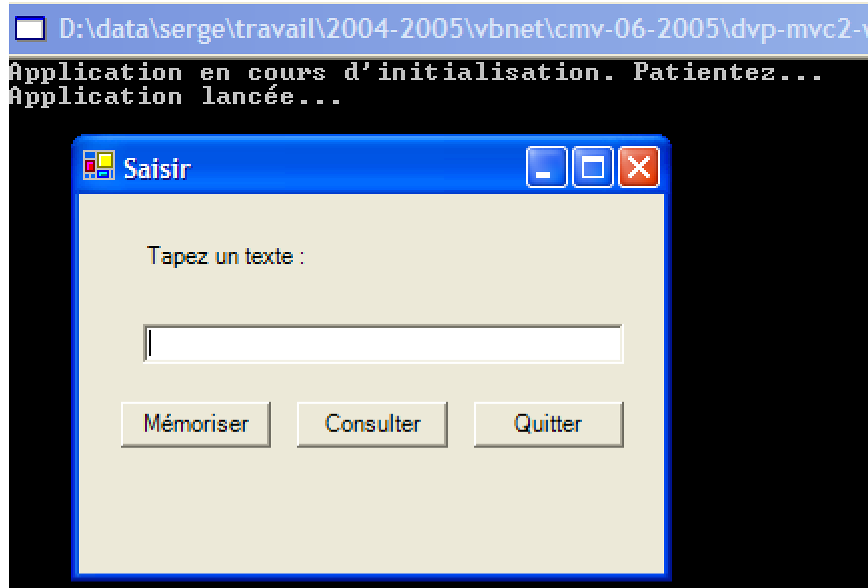
- l'objet [synchro] nécessaire à la synchronisation du contrôleur et des vues est défini lignes 15-19
- l'objet [Session] partagé par les actions et les vues est défini ligne 21. Il sera injecté dans chaque vue et chaque action.
- les vues [VueSaisir, VueConsulter, VueErreurs] sont définies lignes 23-55. On y définit leurs attributs [nom, synchro, session].
- l'unique action [ActionConsulter] est définie lignes 57-61. On y injecte l'objet [Session] défini ligne 21.
- les informations sur les différentes actions du contrôleur sont données lignes 63-82.
- l'objet [infosActionSaisir] sera associé à la première action du contrôleur. Les lignes 78-82 disent que sur cette action, la vue [VueSaisir] doit être affichée.
- l'objet [infosActionConsulter] sera associé à l'action "consulter", celle qui sera déclenchée par le clic sur le bouton [Consulter]. Les lignes 63-77 disent les choses suivantes :
  - lignes 64-66 : que le contrôleur doit commencer par exécuter l'action [actionConsulter] définie lignes 57-61.
  - lignes 67-76 : que sur le résultat "erreurs" de l'action [actionConsulter], la vue [VueErreurs] doit être affichée et que sur le résultat "consulter", la vue [VueConsulter] doit être affichée.
- le contrôleur est défini lignes 84-104. Les attributs [synchro, firstActionName, lastActionName] sont définis lignes 85-93, la liste des actions contrôlées, lignes 94-102.

7.6 Les tests

Une fois générée la solution, nous avons le dossier [bin] suivant :



Un double-clic sur [m2vc-win.exe] lance l'application :



Un clic sur le bouton [Consulter] aboutira de façon aléatoire soit à la vue [VueConsulter], soit à la vue [VueErreurs]. Pour constater ce phénomène, il suffit de cliquer sur le bouton de façon répétée.

8 Conclusion

Nous avons décrit dans ce document un moteur MVC pour des applications windows .NET que nous avons appelé [M2VC-win]. Son architecture est inspirée de celle de Struts, le moteur MVC très utilisé dans le domaine des applications web Java. Nous avons construit trois applications élémentaires utilisant le moteur [M2VC-win]. Seule l'utilisation intensive du moteur dans différentes applications pourrait attester de sa stabilité et de son intérêt. Pour l'instant c'est du béta...

[M2VC-win] est un bon exemple de l'intérêt de [Spring IoC]. Regardons la taille des exécutables de l'application 3 :

Nom	Taille
appli.dll	48 Ko
appli.pdb	48 Ko
log4net.dll	192 Ko
m2vc-win.dll	13 Ko
m2vc-win.exe	8 Ko
m2vc-win.exe.config	4 Ko
Spring.Core.dll	272 Ko

On voit que le moteur [m2vc-win.dll] est peu volumineux (13 K). Le code principal de [Spring] (Spring.Core.dll) occupe lui 272K. C'est l'effet [Spring]. On s'est déchargé sur [Spring] de tout un travail lourd et fastidieux d'instanciations et de gestions de singletons. Cela a conduit à une réduction drastique du code de [M2VC-win] qui se concrétise par le faible poids de sa DLL.

Table des matières

1 INTRODUCTION......2

2 LA PHILOSOPHIE DE STRUTS......3

3 LA PHILOSOPHIE DE M2VC-WIN......4

4 LES ÉLÉMENTS DE M2VC-WIN......5

4.1 L'interface [IControleur]......6
4.2 L'interface [IAction]......6
4.3 L'interface [IVue]......7
4.4 La classe [BaseVue]......7
4.5 La classe [InfosAction]......10
4.6 La classe [BaseControleur]......12
4.7 Le lanceur [m2vc-win.exe]......15
4.8 Le fichier de configuration [m2vc-win.exe.config]......16

5 APPLICATION 1......17

5.1 La structure......18
5.2 Les vues......18
5.3 Le fichier de configuration......19
5.4 Les tests......20

6 APPLICATION 2......21

6.1 La structure......21
6.2 Les vues......22
6.3 Les actions......23
6.4 Le fichier de configuration......23
6.5 Les tests......24

7 APPLICATION 3......25

7.1 La structure de l'application......26
7.2 La session......26
7.3 Les vues......27
7.3.1 La vue [BaseVueAppli]......27
7.3.2 La vue [VueSaisir]......28
7.3.3 La vue [VueConsulter]......29
7.3.4 La vue [VueErreurs]......30
7.4 Les actions......31
7.4.1 L'action [AbstractAction]......31
7.4.2 L'action [ActionConsulter]......31
7.5 Le fichier de configuration......32
7.6 Les tests......33

8 CONCLUSION......34