Migrer de VB6 à VB.NET – Troisième partie

J-M Rabilloud

Remerciements

J'adresse ici tous mes remerciements à l'équipe de rédaction de "developpez.com" et tout particulièrement à Cécile Muno pour le temps qu'ils ont bien voulu passer à la correction et à l'amélioration de cet article.

Introduction

Dans cette partie nous allons parler de composants autonomes, principalement ceux qui ont une interface visuelle : les contrôles. Nous commencerons tout d'abord par étudier l'évolution des formulaires avant de nous lancer dans une étude assez exhaustive de la programmation des contrôles avec VB.NET.

Ce cours considère que vous avez assimilé les deux premières parties, et notamment les concepts d'attribut et d'héritage.

Je vais tenter d'être le plus complet possible, toutefois je pars du principe que vous savez bien manipuler les contrôles en VB 6. Nous verrons un grand nombre d'exemple comme à notre habitude, mais je ne parlerai pas des contrôles dépendants qui ont fait l'objet d'un autre cours.

Bonne lecture.

Formulaire

Généralités

A tout seigneur tout honneur, étudions d'abord les formulaires de VB.NET. Comme je vous l'ai dit dans la première partie, chaque formulaire est maintenant une classe indépendante, qui hérite de System.Windows.Forms.Form. L'espace de nom System.Windows.Forms contient d'ailleurs les classes de formulaires, de contrôles et de certains autres composants visuels.

Fondamentalement le fonctionnement des formulaires est assez différent de VB6, mais la génération de code de l'IDE tend à le masquer. Créons donc un formulaire et regardons le code généré.

```
Public Class Form1
    Inherits System.Windows.Forms.Form

#Region " Code généré par le Concepteur Windows Form "

    Public Sub New()
        MyBase.New()

        'Cet appel est requis par le Concepteur Windows Form.
        InitializeComponent()

        'Ajoutez une initialisation quelconque après l'appel
InitializeComponent()

    End Sub

    'La méthode substituée Dispose du formulaire pour nettoyer la liste
des composants.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Requis par le Concepteur Windows Form
    Private components As System.ComponentModel.IContainer

    'REMARQUE : la procédure suivante est requise par le Concepteur
Windows Form
```

```
    'Elle peut être modifiée en utilisant le Concepteur Windows Form.
    'Ne la modifiez pas en utilisant l'éditeur de code.
    <System.Diagnostics.DebuggerStepThrough()> Private Sub
InitializeComponent()
        '
        'Form1
        '
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
        Me.ClientSize = New System.Drawing.Size(648, 397)
        Me.Name = "Form1"
        Me.Text = "Form1"

    End Sub

#End Region

End Class
```

Que pouvons-nous dire de ce code ?

Comme chaque formulaire est une classe à part entière, le générateur de code :

- Ecrit le code d'un constructeur qui fera appel à la méthode InitializeComponent
- Crée une variable de type Icontainer afin de pouvoir récupérer les contrôles contenus
- Substitue la méthode Dispose du formulaire afin de pouvoir libérer les contrôles contenus
- Crée une méthode d'initialisation

Tout cela est somme toute logique, mais penchons-nous sur un point intéressant : la méthode InitializeComponent. Dans cette méthode, le formulaire va gérer l'ensemble de ses propriétés. Comme il est aussi conteneur, c'est ici que vont se gérer les contrôles contenus.

Pour pouvoir mieux comprendre, j'ajoute une zone de texte (TextBox) dont j'initialise la propriété 'Enabled' à False et un bouton (Button) à mon formulaire. Ma méthode InitializeComponent (générée automatiquement) va devenir :

```
<System.Diagnostics.DebuggerStepThrough()> Private Sub
InitializeComponent()
        Me.TextBox1 = New System.Windows.Forms.TextBox
        Me.Button1 = New System.Windows.Forms.Button
        Me.SuspendLayout()
        '
        'TextBox1
        '
        Me.TextBox1.Location = New System.Drawing.Point(96, 80)
        Me.TextBox1.Name = "TextBox1"
        Me.TextBox1.Size = New System.Drawing.Size(168, 20)
        Me.TextBox1.TabIndex = 0
        Me.TextBox1.Text = "TextBox1"
        '
        'Button1
        '
        Me.TextBox1.Enabled = False
        Me.Button1.Location = New System.Drawing.Point(360, 80)
        Me.Button1.Name = "Button1"
        Me.Button1.Size = New System.Drawing.Size(88, 24)
        Me.Button1.TabIndex = 1
        Me.Button1.Text = "Button1"
        '
        'Form1
        '
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
```

```
        Me.ClientSize = New System.Drawing.Size(648, 397)
        Me.Controls.Add(Me.Button1)
        Me.Controls.Add(Me.TextBox1)
        Me.Name = "Form1"
        Me.Text = "Form1"
        Me.ResumeLayout(False)

    End Sub
```

C'est très proche de VB6 mais, à la différence de celui-ci, vous voyez le code généré. Supposons que je crée un formulaire VB 6, je lui ajoute les deux mêmes contrôles en désactivant aussi la propriété Enabled du TextBox. Je ne vois aucun code dans ma feuille VB6, cependant si j'ouvre mon fichier Form1.frm avec un éditeur de texte, je lis :

```
VERSION 5.00
Begin VB.Form Form1
  Caption        =   "Form1"
  ClientHeight   =   7755
  ClientLeft     =   60
  ClientTop      =   345
  ClientWidth    =   9915
  LinkTopic      =   "Form1"
  ScaleHeight    =   7755
  ScaleWidth     =   9915
  StartUpPosition =   3  'Windows Default
  Begin VB.CommandButton Command1
    Caption        =   "Command1"
    Height         =   495
    Left           =   5760
    TabIndex       =   1
    Top            =   480
    Width          =   2535
  End
  Begin VB.TextBox Text1
    Enabled        =   0   'False
    Height         =   495
    Left           =   840
    TabIndex       =   0
    Text           =   "Text1"
    Top            =   720
    Width          =   1815
  End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Option Explicit
```

Ceci est bien identique. La seule différence réside dans l'accessibilité du code généré. Par contre, vous voyez que tout ce qui touche au positionnement et aux dimensions a changé de syntaxe.

Autrement formulé, la principale différence revient à dire que les formulaires VB6 étaient presque des objets alors que les formulaires VB.NET, eux, sont purement des classes. Vous pouvez dériver des formulaires de vos propres formulaires, les règles de portée des membres s'appliquent normalement.

De même, vous pouvez ajouter des classes dans le fichier du formulaire si vous en avez besoin.

Modifications & nouveautés

Peu de choses ont véritablement changé, nous allons donc rapidement survoler les nouveaux points. Tout d'abord, essayez d'harmoniser vos noms entre la propriété Name du formulaire et le nom du fichier de classe. J'avoue que je ne vois pas pourquoi les deux ne se suivent pas automatiquement. Vous noterez aussi que la propriété 'Caption' a disparu. Pour des raisons d'harmonisation, le texte affiché par une feuille, mais aussi par les contrôles, se retrouve maintenant toujours dans la propriété 'Text'.

Position de démarrage

Pour définir la position de démarrage d'un formulaire en VB6, on utilisait une des constantes de la propriété 'Startup' ou, pour des cas plus exotiques, on réglait celle-ci avec la fenêtre de présentation des formulaires. Maintenant, on utilise une des constantes de la propriété 'StartPosition', telle que :

Nom de membre	Description
CenterParent	Le formulaire est centré dans la position de son formulaire parent.
CenterScreen	Le formulaire est centré dans l'écran et possède les dimensions spécifiées par sa propriété Size.
Manual	La position du formulaire est déterminée par sa propriété Location, sa taille par sa propriété Size.
WindowsDefaultBounds	Le formulaire est situé à l'emplacement par défaut de Windows et possède la taille déterminée par défaut par Windows.
WindowsDefaultLocation	Le formulaire est situé à l'emplacement par défaut de Windows et possède les dimensions spécifiées par sa propriété Size.

Ordre de tabulation

Vous pouvez maintenant gérer aisément les TabIndex de vos contrôles en mode visuel dans l'IDE. Cela se trouve dans le menu Affichage - *Ordre de tabulation*. Cependant, il demeure possible de procéder comme avec VB6.

Formulaire défilant

Il s'agit là d'une évolution importante. Si la notion de zone cliente existe toujours, elle offre maintenant des possibilités supplémentaires. Pour mémoire, la zone cliente d'un contrôle conteneur est la surface de celui-ci moins la barre de titre, les bordures, la barre de menu et les éventuelles barres de défilement.

Dans VB 6, la zone cliente était forcément visible dans le formulaire. Lorsqu'on souhaitait outrepasser celle-ci, on mettait un contrôle conteneur dans la feuille, on lui donnait les dimensions désirées, puis on ajoutait des contrôles barre de défilement pour faire scroller le contrôle.

Dans VB.NET, la zone cliente peut être plus grande que la partie visible du formulaire. Envisageons le cas suivant :

```
Private Sub frmNouv_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim MonBouton As New Windows.Forms.Button
    With MonBouton
        .Name = "cmdScroll"
        .Size = New Size(88, 24)
        .Location = New Point(Me.Height + 300, Me.Width + 300)
        .Text = "Dummy"
    End With
    Me.Controls.Add(MonBouton)
    Me.AutoScroll = True
End Sub
```

Dans cet exemple, je crée un nouveau bouton de commande que je positionne au-delà des limites du formulaire. Pour pouvoir l'atteindre, j'initialise à vrai la propriété 'AutoScroll'. À l'exécution, vous voyez que les barres de défilement sont automatiquement affichées, permettant d'accéder au bouton.

Bien évidemment, vous pouvez utiliser la zone client en mode design. Pour cela, ajouter votre contrôle au formulaire, puis modifier sa propriété 'Location' à celle désirée. Si la propriété 'AutoScroll' est vrai, vous voyez s'afficher les barres de défilements.

Divers

Vous pouvez maintenant gérer l'opacité par la propriété 'Opacity'. Il est aussi possible de forcer le formulaire à rester celui du premier plan en utilisant sa propriété TopMost. Enfin, en jouant avec les propriétés TransparencyKey et des BackColor communs, il y a moyen de définir des zones permettant d'envoyer directement les commandes clavier / souris vers le formulaire parent.

Boite de dialogues

On peut définir une boite de dialogue comme étant un formulaire modal qui renvoie une ou plusieurs informations vers l'appelant. Un formulaire modal est un formulaire qui bloque l'exécution du code appelant jusqu'à qu'il soit masqué ou déchargé. En VB6, le transfert des informations de la boite de dialogue vers l'appelant pouvait être plus ou moins alambiqué selon la complexité des données à renvoyer. On pouvait utiliser une ou plusieurs variables globales, une fonction, une structure, une classe ou une combinaison de tout ceci, bref un peu tout ce qu'on voulait. Un formulaire modal dans VB6 était obtenu en passant le paramètre vbModal à la méthode Show de l'objet Form.

Dans VB.NET, on utilise la méthode surchargée ShowDialog de l'objet Form qui suit les syntaxes suivantes :

Overloads Public Function ShowDialog() As DialogResult
Overloads Public Function ShowDialog(ByVal *owner* **As IWin32Window) As DialogResult**

Où *owner* peut être tout objet implémentant l'interface Iwin32Window (Handle HWND Windows)

Et « DialogResult » pouvant prendre les valeurs :

Nom de membre	Description
Abort	Equivalent au bouton MsgBox 'Abandonner'
Cancel	Equivalent au bouton MsgBox 'Annuler'
Ignore	Equivalent au bouton MsgBox 'Ignorer'
No	Equivalent au bouton MsgBox 'Non'
None	Nothing est retourné à partir de la boîte de dialogue. Cela signifie que l'exécution de la boîte de dialogue modale se poursuit.
OK	Equivalent au bouton MsgBox 'OK'
Retry	Equivalent au bouton MsgBox 'Réessayer'
Yes	Equivalent au bouton MsgBox 'Oui'

N'allez pas croire que seules ces valeurs peuvent être renvoyées à l'appelant. Il s'agit de valeurs pouvant être récupérées par l'appelant directement sur l'appel de ShowDialog, mais comme notre formulaire est maintenant une classe à part entière, il ne subsiste aucun problème de récupération.

 Notez bien que l'appui sur la croix de fermeture d'un formulaire modal revient à le masquer et ne déclenche pas un appel de sa méthode Close comme dans le cas d'un formulaire normal.

Dès lors :

- Vous n'avez pas à instancier de nouveau pour afficher la boite tant que vous ne l'avez pas explicitement détruite
- Vous devez appeler la méthode Dispose pour détruire le formulaire

Exemple

Dans cet exemple, nous allons construire une boite de dialogue de saisie d'éléments pour une collection 'Personne'.

Commençons par réviser notre programmation VB.NET en créant la classe personne et la collection fortement typée :

```
Public Class Personne
    Private pNom As String
    Private pPrenom As String
    Private pDateNaissance As Date

    Public Sub New()

    End Sub

    Public Sub New(ByVal LeNom As String, ByVal LePrenom As String, ByVal
LaDateNaissance As Date)
        Nom = LeNom
        Prenom = LePrenom
        DateNaissance = LaDateNaissance
    End Sub

    Public Property Nom() As String
        Get
            Return pNom
        End Get
        Set(ByVal Value As String)
            pNom = Value
        End Set
    End Property

    Public Property Prenom() As String
        Get
            Return pPrenom
        End Get
        Set(ByVal Value As String)
            pPrenom = Value
        End Set
    End Property

    Public Property DateNaissance() As Date
        Get
            Return pDateNaissance
        End Get
        Set(ByVal Value As Date)
            pDateNaissance = Value
        End Set
    End Property

End Class

Public Class CollPersonne
    Inherits CollectionBase
```

```
    Public Sub New()
        MyBase.New()
    End Sub

    Public Sub New(ByVal UnePersonne As Personne)
        MyBase.New()
        Me.Add(UnePersonne)
    End Sub

    Public Sub Add(ByVal UnePersonne As Personne)
        MyBase.List.Add(UnePersonne)
    End Sub

    Public ReadOnly Property Item(ByVal Index As Int32) As Personne
        Get
            Return CType(MyBase.List.Item(Index), Personne)
        End Get
    End Property
End Class
```

Rien que nous n'ayons déjà vu dans les cours précédents. Je crée maintenant un formulaire principal (frmMain) dans lequel je place un bouton pour ajouter des personnes à ma collection.

Je crée enfin ma boite de dialogue de saisie avec deux zones de texte pour le nom et le prénom et un DateTimePicker pour la date de naissance. Je place aussi un bouton OK et un bouton Annuler.

Le code de mon formulaire est :

```
<Flags()> Private Enum Validation As Integer
    Nom = 1
    Prenom = 2
    Naissance = 4
End Enum

Private MaValid As Validation

Private Sub cmdValider_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles cmdValider.Click
    If txtNom.Text.Trim.Length > 0 AndAlso
System.Text.RegularExpressions.Regex.IsMatch(txtNom.Text, "^[a-zA-Z]+([a-
zA-Z])?(\')?([a-zA-Z]+)?(\-)?([a-zA-Z]+)?(\ )?([a-zA-Z]+)?$") Then
        MaValid = MaValid Or Validation.Nom
    End If
    If txtPrenom.Text.Trim.Length > 0 AndAlso
System.Text.RegularExpressions.Regex.IsMatch(txtPrenom.Text, "^[a-zA-
Z]+([a-zA-Z])?(\')?([a-zA-Z]+)?(\-)?([a-zA-Z]+)?(\ )?([a-zA-Z]+)?$") Then
        MaValid = MaValid Or Validation.Prenom
    End If
    If Me.dtpDateNaissance.Value.CompareTo(Date.Now) <= 0 Then
        MaValid = MaValid Or Validation.Naissance
    End If
    If MaValid = 7 Then
        Me.cmdValider.DialogResult = DialogResult.OK
    End If
End Sub
```

Je mets aussi la propriété DialogResult de mon bouton annuler à Cancel.

 Ne surtout pas mettre la propriété DialogResult du bouton valider sur OK.

En effet, dès que la propriété DialogResult est différente de 'None', un click sur le bouton entraînera le masquage du formulaire et donc la poursuite du code appelant. C'est pour cela que j'affecte la valeur OK par le code.

Enfin le code appelant de ma fenêtre principale est :

```
Private Sub cmdAjouter_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles cmdAjouter.Click
    Dim MaBoite As New frmSaisie
    If MaColPersonne Is Nothing Then MaColPersonne = New CollPersonne
    If MaBoite.ShowDialog(Me) = DialogResult.OK Then
        MaColPersonne.Add(New Personne(MaBoite.txtNom.Text,
MaBoite.txtPrenom.Text, MaBoite.dtpDateNaissance.Value))
        MaBoite.Close()
        MaBoite.Dispose()
    End If
End Sub
```

Discutons-le ensemble. Je crée une instance de ma boite de dialogue ainsi qu'une instance de ma collection si celle-ci n'existe pas. J'appelle ma fenêtre comme une boite de dialogue à l'aide de sa méthode ShowDialog. Comme le formulaire est modal, c'est sur cette ligne qu'aura lieu l'arrêt du code appelant. Je teste la valeur DialogResult retournée. Dans ce cas, c'est soit Cancel si j'ai appuyé sur le bouton annuler, soit OK si la validation est réussie. Notez au passage que si je ferme la boite de dialogue avec la croix de fermeture, c'est Cancel qui sera renvoyé. Si c'est OK qui est renvoyé, j'ajoute un élément à ma collection avec les valeurs récupérées dans la boite de dialogue.

Nous verrons un cas plus complexe à la fin de ce cours.

Evolution des contrôles

Généralités

Les contrôles VB 6 se retrouvent à peu près tous dans VB.NET sous une forme ou sous une autre, à l'exception notable du Conteneur OLE. Nous allons voir qu'on trouve souvent de petites différences dans leur manipulation, mais l'habitude est vite prise puisque les changements sont assez logiques. Par exemple vous ne trouverez plus de propriété Caption puisque maintenant le texte d'un contrôle se retrouve toujours dans la propriété Text.

Dans certains cas, des contrôles ont disparu au profit de classe, dans d'autres cas, l'approche est différente. Par exemple il existe maintenant un contrôle ToolTip que l'on utilise en lieu et place des propriétés ToolTip de VB6.

Evènements

Les contrôles VB.NET gèrent beaucoup plus d'évènements que leurs homologues VB6. Pour vous donner un exemple, regardez l'évolution de la liste des évènements du TextBox

Evénement TextBox VB.NET			
AcceptsTabChanged	**AutoSizeChanged**	BackColorChanged	BorderStyleChanged
BackgroundImageChanged	BindingContextChanged	ChangeUICues	**Click**
CausesValidationChanged	ControlAdded	ControlRemoved	*Disposed*
ContextMenuChanged	CursorChanged	DockChanged	DragDrop
DoubleClick	DragOver	DragEnter	DragLeave
EnabledChanged	Enter	FontChanged	ForeColorChanged
GiveFeedback	GotFocus	HandleCreated	HandleDestroyed
HideSelectionChanged	HelpRequested	ImeModeChanged	Invalidated
KeyDown	KeyPress	KeyUp	Layout
Leave	LocationChanged	LostFocus	**ModifiedChanged**
MouseDown	MouseEnter	MouseHover	MouseLeave
MouseMove	MouseUp	MouseWheel	Move
MultilineChanged	Paint	ParentChanged	QueryContinueDrag
QueryAccessibilityHelp	ReadOnlyChanged	Resize	RightToLeftChanged
SizeChanged	StyleChanged	SystemColorsChanged	TabIndexChanged
TabStopChanged	**TextAlignChanged**	TextChanged	Validated
Validating	VisibleChanged		

Les évènements en gras ne sont pas hérités de Control, c'est-à-dire sont spécifiques à une famille de contrôles. Disposed est fourni par Component

Evénement TextBox VB 6			
Change	Click	DblClick	DragDrop
DragOver	GotFocus	KeyDown	KeyPress
KeyUp	LostFocus	MouseDown	MouseMove
MouseUp	Validate		

Dans cette liste je n'ai pas repris les évènements liés à DDE ou à OLE qui n'ont plus de sens dans VB.NET.

Comme vous le voyez, il y a vraiment une grosse différence. Cependant la plupart de ces évènements arrivent par héritage de la classe Control, c'est-à-dire sont communs à tous les contrôles VB.NET.

Modifications générales

Nous allons d'abord voir les modifications apportées à tous les contrôles, elles sont de loin les plus nombreuses. Comme nous le verrons, les modifications spécifiques sont assez peu nombreuses.

Taille et position

Dans VB6, on gérait la géométrie des contrôles à l'aide des propriétés Left, Top, Height, Width. Les mêmes propriétés existent dans VB.NET, mais on utilise maintenant plutôt une structure *Point* pour le positionnement et une structure *Size* pour les dimensions. Ceci ne change pas grand chose pour les cas simples, mais peut être très utile lors de la gestion de positionnements plus complexes.

Ces structures sont définies dans l'espace de nom System.Drawing que vous allez apprendre à bien connaître puisqu'il est fondamental pour la programmation des contrôles visuels. En tout état de cause, les possibilités graphiques de VB.NET sont largement supérieures à celle de VB6.

Nous allons commencer par un petit exemple, faire une zone de texte qu'on peut déplacer avec la souris à l'exécution.

```
Imports System.Drawing

Public Class Form1

Inherits System.Windows.Forms.Form

Private TextMove As Boolean
    Private MonRect As New Rectangle(0, 0, 0, 0)
    Dim GestGraph As Graphics

    Private Sub TextBox1_MouseDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles TextBox1.MouseDown
        TextMove = True
        Me.TextBox1.Hide()
        MonRect.Size = TextBox1.Size
        MonRect.Location = Me.PointToClient(Me.MousePosition)
        GestGraph = Me.CreateGraphics
    End Sub

    Private Sub Form1_MouseMove(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles MyBase.MouseMove,
TextBox1.MouseMove

        Dim Pinceau As New SolidBrush(Color.White)
        Dim Stylo As New Pen(Color.Black, 1)
        If TextMove AndAlso e.Button = MouseButtons.Left Then
            GestGraph.Clear(Me.BackColor)
            MonRect.Location = Me.PointToClient(Me.MousePosition)
            GestGraph.DrawRectangle(Stylo, MonRect)
            GestGraph.FillRectangle(Pinceau, MonRect)
        End If

    End Sub

    Private Sub Form1_MouseUp(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles MyBase.MouseUp,
TextBox1.MouseUp
        If TextMove Then
```

```
            If MonRect.IntersectsWith(New Rectangle(Button1.Location,
Button1.Size)) Then
                MsgBox("impossible de placer sur le bouton")
            Else
                Me.TextBox1.Location = Me.PointToClient(Me.MousePosition)
            End If
            TextMove = False
            GestGraph.Clear(Me.BackColor)
            Me.TextBox1.Show()
            GestGraph.Dispose()
        End If
    End Sub

End Class
```

Vous allez me dire qu'avec un bon vieux contrôle Shape de VB6, on faisait la même chose avec moins de code. Néanmoins là, je fais un travail purement graphique sans utiliser aucun contrôle supplémentaire.

Je vous disais donc que les possibilités graphiques étaient supérieures. Vérifions-le immédiatement, avec un autre exemple simple. Le code suivant écrit sur la feuille un texte vertical et l'encadre :

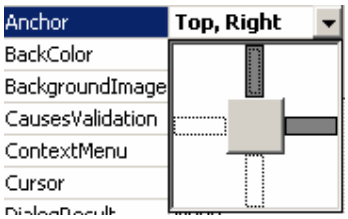
```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click

    Dim GestGraph As System.Drawing.Graphics = Me.CreateGraphics()
    Dim strLegende As String = "Texte vertical"
    Dim MaFont As New System.Drawing.Font("Arial", 12)
    Dim Pinceau As New
System.Drawing.SolidBrush(System.Drawing.Color.Red)
    Dim strFormat As New System.Drawing.StringFormat
    strFormat.FormatFlags = StringFormatFlags.DirectionVertical
    GestGraph.DrawString(strLegende, MaFont, Pinceau, New PointF(100,
100), strFormat)
    GestGraph.DrawRectangle(New Pen(Color.Blue), New Rectangle(100, 100,
GestGraph.MeasureString(strLegende, MaFont).ToSize.Height,
GestGraph.MeasureString(strLegende, MaFont).ToSize.Width))
    MaFont.Dispose()
    Pinceau.Dispose()
    GestGraph.Dispose()

End Sub
```

Vous pouvez noter que je fais des appels explicites aux méthodes Dispose de mes objets graphiques. Je vous ai dit dans la première partie du cours qu'on ne faisait généralement pas d'appel au destructeur. Cependant, lorsque mes objets consomment des ressources non managées, il faut toujours veiller à les libérer explicitement.

Sachez enfin qu'il est possible de passer dans le même temps la taille et la position à l'aide de la méthode SetBounds.

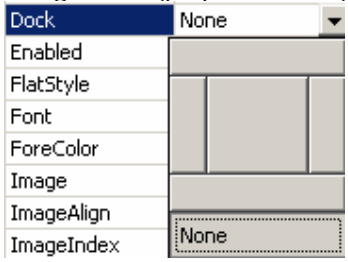
Ancrage et alignement

La plupart des contrôles exposent maintenant une propriété Anchor, qui se présente dans l'éditeur sous la forme :



Vous pouvez dans cette image sélectionner les ancrages désirés. Un contrôle ancré se déplace avec le(s) bord(s) au(x)quel(s) il est ancré lors d'un redimensionnement. Le plus simple pour comprendre consiste à ajouter un contrôle à droite d'un formulaire, d'exécuter puis de redimensionner le coté droit de la fenêtre. Ensuite ancrez le contrôle à droite et recommencez.

L'alignement (propriété 'Dock') se présente sous la forme :



Comme vous le devinez sûrement, ceci aligne le contrôle sur un bord de la fenêtre ou sur toute la zone cliente. Notez que certains contrôles ont une propriété Dock définie dès leur création comme, par exemple, le contrôle StatusBar qui est aligné en bas par défaut.

Apparence

La propriété 'Appearance' VB 6 n'existe plus (celle qui permettait de choisir entre Flat et 3D). Dans VB.NET, la propriété BorderStyle a été modifiée pour englober la combinaison des propriétés BorderStyle et Appearance de VB 6, tel que :

Visual Basic 6.0	Visual Basic .NET
Appearance = 0 – Flat BorderStyle = 0 – None	BorderStyle = BorderStyle.None
Appearance = 0 – Flat BorderStyle = 1 – Fixed Single	BorderStyle = BorderStyle.FixedSingle
Appearance = 1 – 3D BorderStyle = 0 – None	BorderStyle = BorderStyle.None
Appearance = 1 – 3D BorderStyle = 1 – Fixed Single	BorderStyle = BorderStyle.Fixed3D

Pour les boutons de commande, les boutons d'options et les cases à cocher, il fallait auparavant mettre la propriété Style à Graphical pour manipuler des images. Ce n'est plus utile maintenant.

Héritage du parent

Certaines propriétés des contrôles peuvent être héritées de leur parent (dans le sens conteneur) lorsqu'elles ne sont pas explicitement définies. Par exemple, si vous modifiez la police d'un formulaire, la police des contrôles contenus sera modifiée aussi si vous aviez laissé la valeur par défaut.

Visite guidée

Nous allons maintenant voir les modifications intéressantes entre les contrôles VB.NET et VB6 par type de contrôle. Je n'entrerai pas dans les modifications induites par la liaison de données, cela étant plus un aspect des contrôles dépendants.

Label

Ce contrôle n'a pas beaucoup changé. Comme pour tous les autres contrôles sa propriété Caption se retrouve désormais dans la propriété Text. La propriété WordWrap n'existe plus et est toujours vraie dans VB.NET.

Button (CommandButton)

Le contrôle s'utilise de la même façon. Par contre, il n'y a plus de propriété DownPicture. Pour simuler celle-ci, vous devez gérer la propriété Image du bouton par le code.

Par ailleurs, un bouton de commande ne peut plus être 'Default' ou 'Cancel'. Ces boutons particuliers se définissent au niveau du formulaire, dans les propriétés 'CancelButton' et 'AcceptButton'.

CheckBox

Comme pour le cas des boutons de commande, il n'y a plus de propriété DownPicture et DisablePicture.

L'évènement de changement de valeur n'est plus Click mais CheckStateChanged. La propriété Alignement s'appelle maintenant CheckAlign.

La propriété AutoCheck valide la modification de valeur du contrôle par un clic de souris. Si vous la mettez à faux, vous devez gérer les changements de valeur du contrôle par le code.

RadioButton (OptionButton)

Pas de modification significative. Le changement d'état est maintenant l'évènement CheckedChange. Les mêmes remarques sont applicables que pour les cases à cocher.

ListBox

Nous entrons là dans un contrôle dont la manipulation a changé.

Manipulation des éléments

Les éléments d'une zone de liste sont accessibles uniquement par le biais de la collection Items de la liste. Un exemple de manipulation pourrait être :

```
Dim cmpt As Int32
For cmpt = 1 To 100
    Me.ListBox1.Items.Add("élément" & cmpt.ToString)
Next
With Me.ListBox1.Items
    .Remove("élément30")
    .RemoveAt(50)
    .Insert(29, "élément30")
End With
```

Comme il s'agit d'une collection, toutes les propriétés et méthodes d'une collection sont accessibles au travers de la propriété Items.

La sélection des éléments se gère par la propriété SelectionMode assez semblable à la propriété MultiSelect du contrôle VB6 tel que :

VB.NET	VB 6	Description
None		Sélection interdite
One	0	Un seul élément peut être sélectionné.
MultiSimple	1	Plusieurs éléments peuvent être sélectionnés.
MultiExtended	2	Plusieurs éléments peuvent être sélectionnés et l'utilisateur peut utiliser les touches MAJ et CTRL, ainsi que les touches de direction, pour effectuer les sélections.

Autant dans VB6 il fallait toujours passer par les indices pour récupérer les éléments sélectionnés, autant maintenant vous pouvez travailler directement sur les éléments. Le contrôle ListBox VB.NET expose des propriétés SelectedIndex et SelectedItem pour un élément unique, et SelectedItems et SelectedIndices pour une collection lors des sélections multiples.

 Il s'agit de renvoi de collection, elle doit être manipulée comme telle.

Par exemple, le code suivant remplit à nouveau la liste avec les éléments sélectionnés de la liste d'origine :

```
Dim MonEnum As IEnumerator = Me.ListBox1.SelectedItems.GetEnumerator
Dim MaList As ListBox.ObjectCollection = New
ListBox.ObjectCollection(Me.ListBox1)
While MonEnum.MoveNext
    MaList.Add(MonEnum.Current)
End While
Me.ListBox1.Items.Clear()
Me.ListBox1.Items.AddRange(MaList)
```

Notez que vous pouvez désélectionner les éléments en appelant la méthode ClearSelected

Manipulation du contrôle

La propriété MultiColumn détermine la façon dont la liste positionne les éléments s'ils sont plus nombreux que la hauteur de la liste. Si MultiColumn est faux, les éléments sont les uns en dessous des autres et une barre de défilement verticale est ajoutée ; sinon les éléments sont affichés en colonne et une barre de défilement horizontale est éventuellement ajoutée.

Le contrôle présente maintenant une fonctionnalité de recherche avec les méthodes FindString et FindStringExact.

Le code suivant sélectionne tous les éléments de la liste qui commence par "élément1"

```
For cmpt As Int32 = 1 To 100
    Me.ListBox1.Items.Add("élément" & cmpt.ToString)
Next
cmpt = -1
Do
    cmpt = ListBox1.FindString("élément1", cmpt)
    If cmpt <> -1 Then
        If ListBox1.SelectedIndices.Count > 0 Then
            If cmpt = ListBox1.SelectedIndices(0) Then
                Exit Do
            End If
        End If
        ' utilise SetSelected pour forcer la sélection
        ListBox1.SetSelected(cmpt, True)
    End If
Loop While cmpt <> -1 AndAlso cmpt < Me.ListBox1.Items.Count - 1
Me.ListBox1.MultiColumn = True
```

Rem : L'évènement de travail n'est plus Click mais SelectedIndexChanged.

TextBox

La manipulation du contrôle TextBox n'a pas véritablement changé depuis VB6. L'évènement Change s'appelle désormais TextChanged.

 Il est aussi possible de lui affecter par le code un texte de longueur supérieure à MaxLength.

Nous allons quand même voir quelques exemples de manipulation.

Multi lignes

La possibilité de gérer des zones de texte multilignes est régie par plusieurs propriétés.

Multiline ➔ Doit être vrai

WordWrap ➔ Si WordWrap est vrai, le texte revient automatiquement à la ligne quand il atteint le bord droit du contrôle, sinon la barre de défilement horizontale apparaît et le texte continue sur la même ligne

AcceptReturn ➔ Si elle est vraie, un caractère CRLF est envoyé vers la zone de texte, sinon il y activation du bouton défini dans la propriété AcceptButton du formulaire. Notez que si multiline vaut False, la propriété AcceptReturn est toujours considérée comme fausse.

AcceptTab ➔ Si elle est vraie, un caractère TAB est envoyé vers la zone de texte, sinon le contrôle suivant dans l'ordre de TabIndex reçoit le focus.

Imaginons le code suivant :

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    Dim NbLigne As Int32 = Me.TextBox1.Text.Split(New Char()
{Chr(13)}).GetUpperBound(0) + 1
    MsgBox(NbLigne.ToString)
End Sub

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    With Me.TextBox1
        .Multiline = True
        .WordWrap = True
        .AcceptsReturn = True
        .AcceptsTab = False
    End With
End Sub
```

Dans ce code, les retours à la ligne dus à la propriété WordWrap ne sont pas comptabilisés.

Sélection & point d'insertion

C'est souvent un passage mal maîtrisé. On peut gérer par le code la position du point d'insertion et le texte sélectionné. Pour cela, on utilise les propriétés SelectionStart et SelectionLength. La position passée à SelectionStart correspond à un curseur à droite de la position. Donc 0 met le curseur tout à gauche, 1 met le curseur à droite du premier caractère, etc…

SelectionLength est un nombre de caractères. Le code suivant sélectionne tout le texte sauf le dernier caractère :

```
Private Sub TextBox1_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles TextBox1.Click
    Me.TextBox1.SelectionStart = 1
    If TextBox1.Text.Length > 0 Then TextBox1.SelectionLength =
Me.TextBox1.Text.Length - 1
End Sub
```

Il existe aussi une propriété SelectedText qui renvoie le texte sélectionné. Une erreur classique consiste à tenter de l'utiliser pour sélectionner du texte. Supposons que mon contrôle contienne le texte "texte". Si j'écris `Me.TextBox1.SelectedText = "text"` je ne vais pas sélectionner 'text', je vais remplacer la sélection actuelle par 'text'. S'il n'y a pas de texte sélectionné, 'text' va être inséré au niveau du curseur.

ComboBox

Les zones de liste déroulante sont l'union d'une zone de liste et d'une zone de texte, aussi allons nous y retrouver beaucoup de propriétés que nous venons de voir. Cependant, il y a des différences. Par exemple, un ComboBox n'accepte pas de sélections multiples. Il n'y a donc plus de propriétés renvoyant de collection d'éléments sélectionnés.

Regardons maintenant quelques manipulations classiques.

Ajuster la largeur de liste

Le code suivant gère indépendamment la largeur de la liste de la largeur du contrôle pour que tous les éléments s'affichent entièrement :

```
Private Sub ComboBox1_DropDown(ByVal sender As Object, ByVal e As
System.EventArgs) Handles ComboBox1.DropDown
    Dim MonEnum As IEnumerator = Me.ComboBox1.Items.GetEnumerator
    Dim g As Drawing.Graphics = Me.ComboBox1.CreateGraphics
    Dim Taille, MaxTaille As Single
    While MonEnum.MoveNext
        Taille = g.MeasureString(CType(MonEnum.Current, String),
Me.ComboBox1.Font).Width
        If MaxTaille < Taille Then MaxTaille = Taille
    End While
    g.Dispose()
    Me.ComboBox1.DropDownWidth = CInt(MaxTaille) + 20
End Sub
```

Notez que ce n'est pas la meilleure façon de procéder. Nous en verrons une autre plus loin dans ce cours.

Ajouter & supprimer les éléments saisis à la liste

Cette méthode est assez similaire à ce que l'on faisait dans VB6, bien que plus simple du fait de l'existence de la collection Items.

```
Private Sub ComboBox1_KeyDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyEventArgs) Handles ComboBox1.KeyDown
    If e.KeyCode = Keys.Delete Then
        If ComboBox1.Items.Contains(Me.ComboBox1.Text) Then
            Me.ComboBox1.Items.Remove(Me.ComboBox1.Text)
        End If
    ElseIf e.KeyCode = Keys.Enter AndAlso Not
ComboBox1.Items.Contains(Me.ComboBox1.Text) Then
        Me.ComboBox1.Items.Add(Me.ComboBox1.Text)
    End If
End Sub
```

AutoComplete

Par contre, dans ce cas, c'est beaucoup plus simple puisque la méthode FindString permet de générer un code beaucoup plus simple.

```
Private Sub ComboBox1_TextChanged(ByVal sender As Object, ByVal e As
System.EventArgs) Handles ComboBox1.TextChanged
    If Me.ComboBox1.FindString(Me.ComboBox1.Text) > 0 Then
        Dim Pos As Int32 = Me.ComboBox1.Text.Length
        Me.ComboBox1.SelectedIndex =
Me.ComboBox1.FindString(Me.ComboBox1.Text)
```

```
        Me.ComboBox1.SelectionStart = Pos
        Me.ComboBox1.SelectionLength = Me.ComboBox1.Text.Length - Pos
    End If
End Sub
```

PictureBox

Le contrôle PictureBox VB.NET n'est plus un contrôle conteneur. Il regroupe maintenant les fonctionnalités des contrôles PictureBox et Image VB6. Comme nous l'avons vu, les méthodes graphiques ayant changé, il y a de nombreuses modifications, que nous pouvons résumer dans le tableau suivant :

Visual Basic 6.0	Équivalent Visual Basic .NET
Propriété AutoRedraw	Pour assurer la persistance des graphismes, placez les méthodes graphiques dans l'évènement Paint.
Propriété AutoSize (pb) & Strech (Im)	PictureBoxSizeMode.Normal PictureBoxSizeMode.Stretch PictureBoxSizeMode.AutoSize
Méthode Circle	Graphics.DrawEllipse
Propriété ClipControls	Pour ne redessiner qu'une partie d'un objet, créez un clip dans l'évènement Paint à l'aide de la méthode Graphics.SetClip.
Méthode Cls	Graphics.Clear
Propriété CurrentX	Plus de notion de position courante.
Propriété CurrentY	Plus de notion de position courante.
Propriété DrawMode	Pen.Color, propriété
Propriété DrawStyle	Pen.PenType, propriété
Propriété DrawWidth	Pen.Width, propriété
Propriété FillColor	SolidBrush.Color, propriété
Propriété FillStyle	Pen.Brush, propriété
Propriété HasDC	Les contextes de périphériques ne sont plus nécessaires avec GDI+.
Propriété HDC	Les contextes de périphériques ne sont plus nécessaires avec GDI+.
Propriété Image	Retourne un objet System.Drawing.Image. Eqv de la prop. Picture
Méthode Line	Graphics.DrawLine
Méthode PaintPicture	Graphics.DrawImage
Méthode Point	Pas d'équivalent pour les formulaires ou les contrôles. Pour les images bitmap, utilisez la méthode Bitmap.GetPixel.
Méthode Print	Graphics.DrawString
Méthode Pset	Pas d'équivalent pour les formulaires ou les contrôles. Pour les images bitmap, utilisez la méthode Bitmap.SetPixel.
Propriété TextHeight	Font.Height, propriété
Propriété TextWidth	Graphics.MeasureString

Frame (Panel & GroupBox)

Le contrôle Frame, qui était uniquement un contrôle conteneur de regroupement existe maintenant sous forme de deux contrôles. En fait, ces deux contrôles permettent aussi de prendre en charge ce que l'on faisait lorsqu'on utilisait le contrôle PictureBox comme conteneur.

Le contrôle GroupBox possède une bordure et une légende. Le contrôle Panel est défilant, sans bordure et sans légende. Les contrôles conteneur gèrent une collection des contrôles qu'ils contiennent, ceux-ci n'étant pas membres de la collection Controls de la feuille. On peut cependant directement qualifier les contrôles contenus en partant de la feuille. Ceci veut dire qu'il faut faire attention lors de la qualification sous peine d'erreur.

Supposons que je travaille avec trois boutons radio dans un contrôle Panel. Le code généré par Visual Studio est :

```
'Panel1
'
Me.Panel1.Controls.Add(Me.RadioButton3)
Me.Panel1.Controls.Add(Me.RadioButton2)
Me.Panel1.Controls.Add(Me.RadioButton1)
Me.Panel1.Location = New System.Drawing.Point(40, 128)
Me.Panel1.Name = "Panel1"
Me.Panel1.Size = New System.Drawing.Size(168, 200)
Me.Panel1.TabIndex = 2
'
'RadioButton3
'
Me.RadioButton3.Location = New System.Drawing.Point(24, 152)
Me.RadioButton3.Name = "RadioButton3"
Me.RadioButton3.TabIndex = 2
Me.RadioButton3.Text = "RadioButton3"
'
'RadioButton2
'
Me.RadioButton2.Checked = True
Me.RadioButton2.Location = New System.Drawing.Point(24, 88)
Me.RadioButton2.Name = "RadioButton2"
Me.RadioButton2.TabIndex = 1
Me.RadioButton2.TabStop = True
Me.RadioButton2.Text = "RadioButton2"
'
'RadioButton1
'
Me.RadioButton1.Location = New System.Drawing.Point(24, 32)
Me.RadioButton1.Name = "RadioButton1"
Me.RadioButton1.TabIndex = 0
Me.RadioButton1.Text = "RadioButton1"
'
'Form1
'
Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
Me.ClientSize = New System.Drawing.Size(576, 405)
Me.Controls.Add(Me.Panel1)
Me.Controls.Add(Me.ComboBox1)
Me.Controls.Add(Me.Button1)
Me.Name = "Form1"
Me.Text = "Form1"
Me.Panel1.ResumeLayout(False)
```

Si dans mon code j'écris :

```
MsgBox(Me.Controls.IndexOf(Me.RadioButton2).ToString)
```

J'obtiendrais la valeur –1 car RadioButton2 n'est pas membre de la collection Controls du formulaire. Par contre la qualification Me.RadioButton2 est correcte.

Tout cela pour dire qu'il faut faire bien attention à la manipulation des collections Controls, car c'est par leur biais que se définit le parent et que, par définition, il n'y a qu'un parent. Pour illustrer ceci, regardons le code suivant.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click

        Dim RadioButton4 As New Windows.Forms.RadioButton
        With RadioButton4
```

```
            .Location = New System.Drawing.Point(16, 80)
            .Name = "RadioButton4"
            .Text = "RadioButton4"
            .TabIndex = 4
        End With
        Me.Panel1.Controls.Add(RadioButton4)
        Me.Panel1.AutoScroll = True
        'Me.Controls.Add(RadioButton4)

    End Sub
```

En l'état, j'ajoute un bouton radio à mon panel que je rends défilant pour y accéder. Si j'enlève le commentaire de la dernière ligne, je vais voir que mon bouton ne sera plus dans le panel. En effet, le système de coordonnées de la propriété Location est toujours en fonction du parent. En affectant le contrôle à la collection Controls de la feuille, j'ai modifié son parent donc le contrôle se retrouve à la position 16,80 par rapport au coin en haut à gauche de la zone cliente de la feuille et non plus à celui du panel.

Notez enfin que les contrôles conteneurs gèrent l'exclusivité des boutons radio qu'ils contiennent, c'est-à-dire qu'un seul bouton radio peut être sélectionné au sein d'un contrôle conteneur.

Timer

Le contrôle Timer fonctionne comme en VB6 sauf pour la gestion de la propriété Interval. Avec VB.NET il n'est plus possible d'affecter 0 à la propriété Interval pour arrêter le Timer. Vous devez soit mettre la propriété Enabled à False, soit utiliser la méthode Stop. Notez aussi que l'évènement Timer s'appelle désormais Tick.

Groupe de contrôles

La notion de groupes de contrôles (appelés parfois tableau de contrôles) dans le sens VB6 du terme n'existe plus. Elle n'est plus nécessaire du fait de la nouvelle gestion des évènements. En effet, l'intérêt principal des groupes de contrôles VB6 était la possibilité de gérer dans un seul code évènement, les évènements de contrôles connexes. Comme en VB.NET on s'abonne à des évènements, il n'est plus utile de passer par des tableaux.

Dans mon formulaire, je positionne quatre zones de texte en leur laissant leurs noms par défaut (de TextBox1 à TextBox4). Je peux gérer un évènement commun KeyPress ayant la structure suivante :

```
Private Sub TextBox1_KeyPress(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TextBox1.KeyPress,
TextBox2.KeyPress, TextBox3.KeyPress, TextBox4.KeyPress
    Select Case CInt(Microsoft.VisualBasic.Right(CType(sender,
Control).Name, 1))
        Case 1

        Case 2

        Case 3

        Case 4
    End Select
End Sub
```

Les notations de types CInt(Microsoft.VisualBasic.Right(CType(sender, Control).Name, 1)) n'étant pas particulièrement lisible, il est souvent plus rentable de travailler avec les propriétés Tag ou TabIndex des contrôles. En utilisant les instructions AddHandler et RemoveHandler, il est aussi possible de regrouper et de dégrouper les contrôles soumis à une même procédure d'évènements, ce qui est bien plus puissant que la gestion par tableau de contrôles.

Enfin, il demeure toujours possible de regrouper des contrôles par des collections ou des tableaux afin de pratiquer des traitements groupés. Par exemple :

```
Private TabControl(3) As Control

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        TabControl(0) = Me.TextBox1
        TabControl(1) = Me.TextBox2
        TabControl(2) = Me.TextBox3
        TabControl(3) = Me.TextBox4
    End Sub


    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
        Dim cmpt As Int32
        For cmpt = 0 To TabControl.GetUpperBound(0)
            TabControl(cmpt).Visible = False
        Next
    End Sub
```

Nouveaux contrôles

Ce que nous allons voir ici ne sont pas précisément des nouveaux contrôles (bien que certains le soient vraiment) mais des contrôles qui n'étaient pas proposés par défaut dans VB6. Il s'agit de contrôles devant être ajoutés au projet. Je ne vais pas actuellement réaliser une étude détaillée de ces contrôles mais je m'orienterai plutôt vers des exemples d'utilisation classique.

LinkLabel

Il s'agit d'un contrôle label possédant un ou plusieurs lien(s) hypertexte. L'exemple suivant gère un linklabel exposant deux liens.

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim MonLien As New Windows.Forms.LinkLabel
    With MonLien
        .Location = New Point(24, 80)
        .Size = New Size(320, 16)
        'Gestion des couleurs de lien
        .DisabledLinkColor = Color.Gray
        .VisitedLinkColor = Color.Indigo
        .LinkColor = Color.Blue
        'Ajout du texte puis des liens
        .Text = "Vous trouverez de l'aide sur MSDN ou sur Developpez.com"
        .Links.Add(29, 4, "www.microsoft.com/france/vbasic/default.mspx")
        .Links.Add(41, 14, "www.vb.developpez.com")
    End With
    Me.Controls.Add(MonLien)
    AddHandler MonLien.LinkClicked, AddressOf linkLabel_LinkClicked

End Sub

Private Sub linkLabel_LinkClicked(ByVal sender As Object, ByVal e As
System.Windows.Forms.LinkLabelLinkClickedEventArgs)

    'marque le lien cliqué comme visité
    CType(sender, LinkLabel).Links(CType(sender,
LinkLabel).Links.IndexOf(e.Link)).Visited = True
    'Récupère la cible du lien cliqué
    Dim target As String = CType(e.Link.LinkData, String)
    'Démarre Internet Explorer
    If Not target Is Nothing AndAlso target.StartsWith("www") Then
        System.Diagnostics.Process.Start("IExplore.exe", target)
    End If

End Sub
```

ImageList

Ce contrôle sert à fournir des images aux contrôles graphiques de la feuille. Je devrais d'ailleurs dire ce composant, car il ne possède pas d'interface visuelle. La méthode la plus simple pour l'utiliser consiste à ajouter des images en mode Design. Il est cependant assez simple de la gérer par le code. En effet les images sont contenues dans une collection Images qui se manipule comme telle.

Un des avantages évident du travail en mode Design, c'est que les images se chargent dans le contrôle. Dès lors, elles sont disponibles même si vous supprimez les fichiers de l'emplacement d'origine. Si vous travaillez par le code, vous devez embarquer les images avec l'application ou en utilisant des ressources, point que nous étudierons ultérieurement.

Nous pouvons évidemment combiner les deux cas. Dans l'exemple suivant, je vais ajouter une Image à mon composant ImageList et attribuer celle-ci à mon bouton de commande.

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim MonImage As System.Drawing.Image =
Image.FromFile("d:\user\graphics\bitmap\boite.bmp")
    'Me.ImageList1.ImageSize = New Size(64, 64)
    'Dim resources As System.Resources.ResourceManager = New
System.Resources.ResourceManager(GetType(Form1))
    'Me.ImageList1.ImageStream =
CType(Resources.GetObject("ImageList1.ImageStream"),
System.Windows.Forms.ImageListStreamer)
    Me.ImageList1.Images.Add(MonImage)
    MsgBox(Me.ImageList1.Images.Count.ToString)
    With Me.Button1
        .Size = New Size(64, 64)
        .ImageList = Me.ImageList1
        .ImageIndex = Me.ImageList1.Images.Count - 1
        .ImageAlign = ContentAlignment.MiddleCenter
        .Text = ""
    End With
End Sub
```

Comme vous le voyez, la récupération est d'autant plus simple que le contrôle Button expose des propriétés ImageList et ImageIndex.

Je vous ai mis des lignes assez absconses en commentaire.

Le composant ImageList dimensionne toutes les images qu'il contient selon sa propriété ImageSize. Si la valeur de celle-ci change, il recrée un handle pour le ImageListStreamer qu'il utilise (ce qui lui permet d'accéder aux données des images). Vous devez donc réaffecter ce handle, ce que font les lignes en commentaire.

La taille maximale des images est 255*255.

ListView

Un contrôle ListView est un contrôle ListBox amélioré. Globalement, il a deux utilisations :sous forme d'une liste d'objets représentés par des icônes ou sous forme de liste multi colonnes.

Nous allons voir des exemples pour ces deux cas.

Liste d'objet

Il s'agit là d'un exemple assez simple.

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

    With Me.ListView1
        .LargeImageList = Me.ImageList1
        .MultiSelect = False
        .Activation = ItemActivation.TwoClick
        .View = View.LargeIcon
    End With
    For Cmpt As Int32 = 0 To 5
        Me.ListView1.Items.Add("Elément " & Cmpt, Cmpt)
    Next

End Sub
```

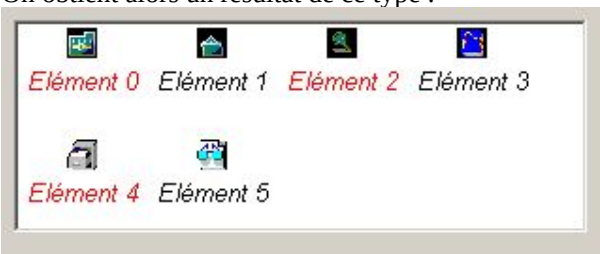
Notez que je passe l'index de l'image souhaitée pour l'élément directement dans le Add. Une écriture plus lisible serait :

```
For Cmpt = 0 To 5
    Dim ObjetList As New ListViewItem
    With ObjetList
        .Text = "Elément " & Cmpt
        .ImageIndex = Cmpt
    End With
    Me.ListView1.Items.Add(ObjetList)
Next
```

Cette écriture est souvent plus intéressante que la première parce qu'elle donne accès à d'autres propriétés de mise en forme des éléments. Ainsi le code suivant met aussi la police en rouge pour un élément sur deux et en italique pour tous les éléments :

```
For Cmpt = 0 To 5
    Dim ObjetList As New ListViewItem
    With ObjetList
        .Text = "Elément " & Cmpt
        .ImageIndex = Cmpt
        If Cmpt Mod 2 = 0 Then .ForeColor = Color.Red
        .Font = New Font("Arial", 10, FontStyle.Italic)
    End With
    Me.ListView1.Items.Add(ObjetList)
Next
```

On obtient alors un résultat de ce type :



Les évènements utilisés pour intercepter la sélection d'un élément, soit SelectedIndexChanged, soit ItemActivate, ne renvoient pas d'arguments particuliers permettant d'identifier l'élément sélectionné. A l'instar du contrôle ListBox, vous devez utiliser SelectedItems ou SelectedIndices pour retrouver les éléments sélectionnés.

Liste multi colonnes

Une liste de ce type est un contrôle ListView dont la propriété View vaut "Details".
L'exemple suivant est une liste de fichier classique ou les fichiers "Zip" seront affichés en vert.

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

    Dim Repertoire As New System.IO.DirectoryInfo("d:\user\tutos")
    Dim ListeFichier() As System.IO.FileInfo = Repertoire.GetFiles
    Me.ListView1.View = View.Details
    With Me.ListView1
        .Columns.Add("Nom", 200, HorizontalAlignment.Center)
        .Columns.Add("Taille", 80, HorizontalAlignment.Right)
        .Columns.Add("Date", 100, HorizontalAlignment.Center)
    End With
    Dim cmpt As Int32
    For cmpt = 0 To ListeFichier.GetUpperBound(0)
        With Me.ListView1.Items.Add(ListeFichier(cmpt).Name)
            .SubItems.Add(ListeFichier(cmpt).Length.ToString)
            .SubItems.Add(ListeFichier(cmpt).CreationTime.ToString)
            If ListeFichier(cmpt).Extension.IndexOf("zip") > 0 Then
                .ForeColor = Color.Green
            End If
        End With
    Next

End Sub
```

TreeView

Le contrôle TreeView est une arborescence similaire à celle de l'explorateur. C'est un contrôle qui présente de nombreuses possibilités, dont nous n'allons voir que les aspects principaux.

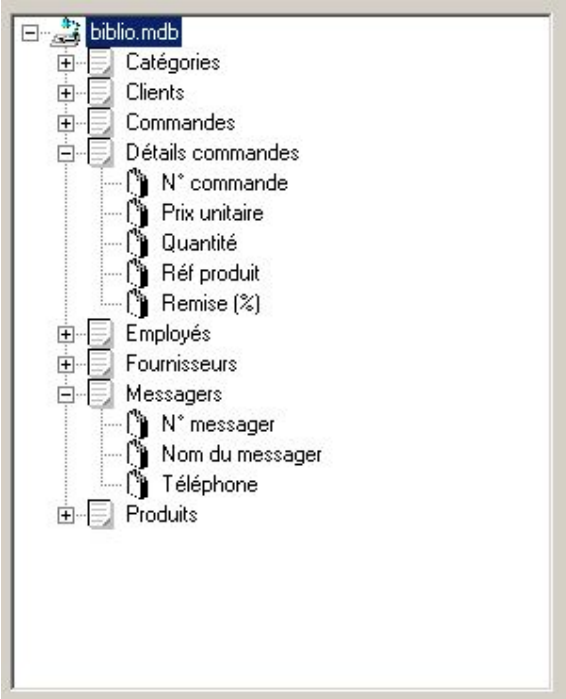
remplissage

Nous allons tout d'abord créer un remplissage d'un TreeView par la structure d'une base de données Access. Je vais le coder un peu lourd pour séparer l'extraction de la structure du remplissage du TreeView sans utiliser une collection fortement typée.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    'code d'extraction de la structure
    Dim lstStructure As New SortedList
    Dim MaConn As New
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\User\NWIND.MDB;")
    MaConn.Open()
    Dim schemaTables As DataTable =
MaConn.GetOleDbSchemaTable(OleDbSchemaGuid.Tables, New Object() {Nothing,
Nothing, Nothing, "TABLE"})
    Dim MonEnumColonne, MonEnumTable As IEnumerator
    Dim schemaColTable As DataTable
    Dim ListColonne As ArrayList
    MonEnumTable = schemaTables.Rows.GetEnumerator
    While MonEnumTable.MoveNext
```

```
        schemaColTable =
MaConn.GetOleDbSchemaTable(OleDbSchemaGuid.Columns, New Object()
{Nothing, Nothing, CType(MonEnumTable.Current,
DataRow).Item("Table_Name"), Nothing})
        ListColonne = New ArrayList
        MonEnumColonne = schemaColTable.Rows.GetEnumerator
        While MonEnumColonne.MoveNext
            ListColonne.Add(CType(MonEnumColonne.Current,
DataRow).Item("column_name"))
        End While
        lstStructure.Add(CType(MonEnumTable.Current,
DataRow).Item("Table_Name"), ListColonne)
    End While
    MaConn.Close()
    'remplissage du treeview
    Dim NodRoot, NodTable, NodTemp As TreeNode
    NodRoot = Me.TreeView1.Nodes.Add("biblio.mdb")
    NodRoot.ImageIndex = 0
    Dim cmpt1, cmpt2 As Int32
    Dim MonEnum As IEnumerator
    For cmpt1 = 0 To lstStructure.Count - 1
        NodTable = NodRoot.Nodes.Add(CType(lstStructure.GetKey(cmpt1),
String))
        MonEnum = CType(lstStructure.GetByIndex(cmpt1),
ArrayList).GetEnumerator
        While MonEnum.MoveNext
            NodTemp = New TreeNode
            With NodTemp
                .Text = CType(MonEnum.Current, String)
                .ImageIndex = 2
            End With
            NodTable.Nodes.Add(NodTemp)
            NodTable.ImageIndex = 1
        End While
    Next
    Me.TreeView1.ImageList = Me.ImageList1

End Sub
```

J'obtiens ainsi un résultat semblable à :



Evènements spécifiques

Il existe une série d'évènements spécifiques, fonctionnant par paire avant/après.

AfterCheck	Se produit après que la case à cocher du nœud d'arbre a été activée.
AfterCollapse	Se produit après la réduction du nœud d'arbre.
AfterExpand	Se produit après le développement du nœud d'arbre.
AfterLabelEdit	Se produit une fois le texte de l'étiquette du nœud d'arbre modifié.
AfterSelect	Se produit après la sélection du nœud d'arbre.
BeforeCheck	Se produit avant que la case à cocher du nœud d'arbre soit activée.
BeforeCollapse	Se produit avant la réduction du nœud d'arbre.
BeforeExpand	Se produit avant le développement du nœud d'arbre.
BeforeLabelEdit	Se produit une fois le texte de l'étiquette du nœud d'arbre modifié.
BeforeSelect	Se produit avant la sélection du nœud d'arbre.

Toutefois, on peut travailler avec des évènements classiques en les adaptant à ce que l'on désire.

Manipulation de l'arborescence

La manipulation de l'arborescence peut être assez complexe si le concept de nœuds n'est pas correctement assimilé.

En fait les objets TreeNodes qui composent un objet TreeView sont des collections récursives. Les règles suivantes s'appliquent toujours :

- Il n'existe qu'un nœud 'Root'. On l'identifie par la valeur « Nothing » de son parent.
- Un nœud n'a toujours qu'un seul parent.
- Un nœud ne fournit des informations que sur ses descendants directs.

Comme il s'agit de collection récursive, le code suivant parcourt tous les nœuds enfants du nœud passé en paramètre :

```
Private Sub ParcoursNoeud(ByVal MonNode As TreeNode)
    MsgBox(MonNode.ToString)
    Dim MonEnum As IEnumerator = MonNode.Nodes.GetEnumerator
    While MonEnum.MoveNext
        ParcoursNoeud(CType(MonEnum.Current, TreeNode))
    End While
End Sub
```

J'utilise bien un code récursif.

La position hiérarchique (parfois appelée niveau d'imbrication) n'est pas toujours simple à connaître. Pour la trouver, je peux utiliser une fonction récursive du type :

```
Private Function Niveau(ByVal MonNode As TreeNode, ByVal Start As
Boolean) As Int32
    Static Pos As Int32
    If Start Then Pos = 0
    If MonNode.Parent Is Nothing Then
        Return Pos
    Else
        Pos += 1
        Niveau(MonNode.Parent, False)
        Return Pos
    End If
End Function
```

Que j'appellerais par exemple avec :

```
MsgBox(Niveau(Me.TreeView1.SelectedNode, True).ToString)
```

Toutefois cette approche n'est souvent pas la meilleure car les procédures récursives sont coûteuses en ressources. Généralement, on connaît à priori le niveau hiérarchique lors de la construction. Il est alors rentable de le stocker dans la propriété Tag de l'objet TreeNode lors de la construction du TreeView.

Pour finir, voyons la fin de notre exemple du début. A l'appel du menu contextuel MenuItem1, je rajoute des informations du champ sélectionné si le nœud sélectionné est bien un champ évidemment.

```
Private Sub MenuItem1_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles MenuItem1.Click
    If Niveau(Me.TreeView1.SelectedNode, True) = 2 Then
        Dim MonReader As OleDbDataReader
        Dim MaConn As New
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\User\NWIND.MDB;")
        Dim strSQL As String = "SELECT [" &
Me.TreeView1.SelectedNode.Text & "] FROM [" &
Me.TreeView1.SelectedNode.Parent.Text & "]"
        Dim MaComm As New OleDbCommand(strSQL, MaConn)
        MaConn.Open()
        MonReader = MaComm.ExecuteReader(CommandBehavior.KeyInfo)
        Dim TableReponse As DataTable = MonReader.GetSchemaTable
        'si la colonne est membre de la clé primaire, met une icone clé
dans l'image du champ
        If CType(TableReponse.Rows(0).Item("IsKey"), Boolean) Then
Me.TreeView1.SelectedNode.ImageIndex = 3
        'si le champ accepte Null, met le noeud du champ en bleu
        If CType(TableReponse.Rows(0).Item("AllowDBNull"), Boolean) Then
Me.TreeView1.SelectedNode.ForeColor = Color.Blue
        'ajoute un sous élément type du champ
```

```
Me.TreeView1.SelectedNode.Nodes.Add(TableReponse.Rows(0).Item("DataType")
.ToString)
    End If
End Sub
```

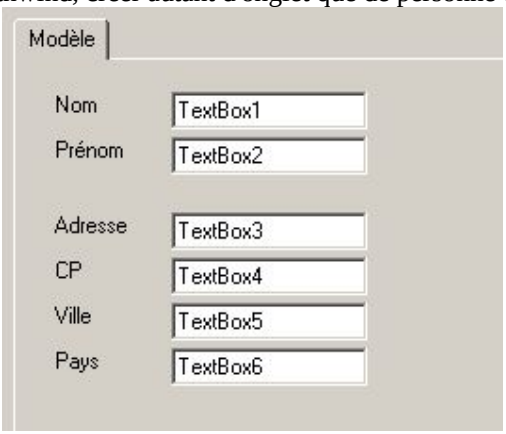
TabControl

Ce contrôle de type multi pages est sensiblement l'équivalent du contrôle SSTab de VB 6.

Il ne fonctionne donc pas intrinsèquement comme le contrôle TabStrip, même si on peut simuler celui-ci.

Chaque onglet est un objet TabPage indépendant. Ces objets sont conteneurs. Chaque onglet peut contenir des objets différents. L'évènement click de l'objet TabPage permet de récupérer l'élément sélectionné. Il n'y a pas tellement de manipulation de ces contrôles par le code habituellement, tout se faisant en mode design, sauf dans un cas : la simulation du TabStrip.

L'intérêt de la logique du TabStrip, c'est de ne pas dupliquer à l'infini les contrôles lorsque les onglets sont connexes. Prenons l'exemple suivant. Je vais parcourir la liste des employés de la base de données Northwind, créer autant d'onglet que de personne et remplir tous les onglets selon le modèle ci-dessous.



Le code est le suivant :

```
Dim MaConn As New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\User\NWIND.MDB;")
Dim MaCommande As New OleDbCommand("SELECT Nom, Prénom, Adresse,[Code
Postal] , ville, Pays FROM Employés", MaConn)
MaConn.Open()
Dim MonReader As OleDbDataReader =
MaCommande.ExecuteReader(CommandBehavior.CloseConnection)
Dim cmpt As Int32 = 0
While MonReader.Read()
    Dim MaPage As New TabPage, MonLabel As Label, MonText As TextBox
    For cmpt = 1 To 6
        MonLabel = New Label
        With MonLabel
            .Text = CType(Choose(cmpt, "Nom", "Prénom", "Adresse", "CP",
"Ville", "Pays"), String)
            .Size = New System.Drawing.Size(56, 16)
            .Location = New Point(16, 16 + (cmpt - 1) * 24)
        End With
        MaPage.Controls.Add(MonLabel)
        MonText = New TextBox
        With MonText
```

```
            .Text = MonReader.GetString(cmpt - 1)
            .Size = New System.Drawing.Size(104, 16)
            .Location = New Point(80, 16 + (cmpt - 1) * 24)
        End With
        MaPage.Text = MonReader.GetString(0) & " " &
MonReader.GetString(1)
        MaPage.Controls.Add(MonText)
    Next
    Me.TabControl1.TabPages.Add(MaPage)
End While
MonReader.Close()
```

Ce code ne pose pas de problème, si ce n'est que je crée 12 contrôles par employés.

L'approche TabStrip consiste à ne créer que des onglets vides, et à travailler sur les douze même objets contenus. Pour faciliter le travail, je vais utiliser une DataTable pour garder une copie locale de mes données.

Le code pourrait être :

```
Private dsEmploye As DataTable

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    Dim MaConn As New
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\User\NWIND.MDB;")
    Dim MaCommande As New OleDbCommand("SELECT Nom, Prénom, Adresse,
[Code Postal],ville , Pays FROM Employés", MaConn)
    Dim MonAdapt As New OleDbDataAdapter(MaCommande)
    dsEmploye = New DataTable
    MonAdapt.Fill(dsEmploye)
    Dim Cmpt As Int32
    For Cmpt = 0 To dsEmploye.Rows.Count - 1
        Me.TabControl1.TabPages.Add(New
TabPage(CType(dsEmploye.Rows(Cmpt).Item(0), String)))
    Next
    Remplissagepage()
End Sub

Private Sub TabControl1_SelectedIndexChanged(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
TabControl1.SelectedIndexChanged
    Remplissagepage()
End Sub

Private Sub Remplissagepage()

    If Me.TabControl1.TabPages.Count = 0 Then Exit Sub
    Dim MaPage As TabPage = Me.TabControl1.SelectedTab
    Dim MonIndex As Int32 = Me.TabControl1.SelectedIndex

    MaPage.Controls.Add(Me.lblNom)
    MaPage.Controls.Add(Me.lblPrenom)
    MaPage.Controls.Add(Me.lblAdresse)
    MaPage.Controls.Add(Me.lblCP)
    MaPage.Controls.Add(Me.lblVille)
    MaPage.Controls.Add(Me.lblPays)
    MaPage.Controls.Add(Me.txtNom)
    MaPage.Controls.Add(Me.txtPrenom)
```

```
    MaPage.Controls.Add(Me.txtAdresse)
    MaPage.Controls.Add(Me.txtCP)
    MaPage.Controls.Add(Me.txtVille)
    MaPage.Controls.Add(Me.txtPays)
    Me.txtNom.Text = CType(dsEmploye.Rows(MonIndex).Item(0), String)
    Me.txtPrenom.Text = CType(dsEmploye.Rows(MonIndex).Item(1), String)
    Me.txtAdresse.Text = CType(dsEmploye.Rows(MonIndex).Item(2), String)
    Me.txtCP.Text = CType(dsEmploye.Rows(MonIndex).Item(3), String)
    Me.txtVille.Text = CType(dsEmploye.Rows(MonIndex).Item(4), String)
    Me.txtPays.Text = CType(dsEmploye.Rows(MonIndex).Item(5), String)
End Sub
```

Dans ce cas, je travaille avec les douze mêmes contrôles pour gérer tous les employés.

DateTimePicker & MonthCalendar

Ces deux contrôles travaillent uniquement sur des dates. Je les regroupe ici car le contrôle DateTimePicker ne nécessite généralement pas ou peu d'action par le code. Celui-ci attend une structure DateTime comme valeur et uniquement cette structure. On peut lui ajouter une case à cocher par sa propriété 'ShowCheckBox'. Par défaut le contrôle présente une flèche de déroulement affichant un calendrier de sélection. On peut préférer une action de type Up/Down en valorisant la propriété ShowUpDown.

Formatage du DateTimePicker

Le contrôle accepte trois formats prédéfinis et permet l'utilisation de chaîne de format personnalisée. Le code suivant modifie le format en fonction de l'option choisie :

```
Private Sub RadioButton_CheckedChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles RadioButton1.CheckedChanged,
RadioButton2.CheckedChanged, RadioButton3.CheckedChanged,
RadioButton4.CheckedChanged, RadioButton5.CheckedChanged
    Select Case CType(sender, RadioButton).Tag
        Case 1
            With Me.DateTimePicker1
                .Value = .MinDate
                .CustomFormat = " "
                .Format = DateTimePickerFormat.Custom
            End With

        Case 2
            Me.DateTimePicker1.Format = DateTimePickerFormat.Long

        Case 3
            Me.DateTimePicker1.Format = DateTimePickerFormat.Short

        Case 4
            Me.DateTimePicker1.Format = DateTimePickerFormat.Time
            Me.DateTimePicker1.Value = DateTime.Now

        Case 5
            DateTimePicker1.CustomFormat = "dd/MM/yy hh:mm:ss"
            DateTimePicker1.Format = DateTimePickerFormat.Custom

    End Select
End Sub
```

Notez que le cas traité dans le 'Case 1' permet de vider l'affichage du contrôle. Cependant celui-ci contient bien toujours une valeur.

MonthCalendar

Ce contrôle est un calendrier. Il affiche un ou plusieurs mois sous forme de tableau bidimensionnel défini dans la propriété CalendarDimensions. Ce contrôle permet la sélection d'une plage de dates, il est aussi possible à l'aide d'un peu de code de sélectionner plusieurs dates discontinues.

Le code suivant met en gras tous les samedis et dimanches et sélectionne la semaine ouvrée suivante par rapport à la date du jour.

```
Dim MaDate, Debut, Fin As Date
Dim NbMois As Int32
With Me.MonthCalendar1
    .TodayDate = Today
    NbMois = .CalendarDimensions.Height * .CalendarDimensions.Width
    Dim cmpt As Int32
    MaDate = .GetDisplayRange(False).Start
    Fin = .GetDisplayRange(False).End
    Do Until MaDate > Fin
        If MaDate.DayOfWeek = DayOfWeek.Saturday OrElse MaDate.DayOfWeek
= DayOfWeek.Sunday Then
            .AddBoldedDate(MaDate)
        End If
        MaDate = MaDate.AddDays(1)
    Loop
    .UpdateBoldedDates()
    MaDate = Today.AddDays(7)
    Select Case MaDate.DayOfWeek
        Case DayOfWeek.Monday
            Debut = MaDate
        Case DayOfWeek.Tuesday
            Debut = MaDate.AddDays(-1)
        Case DayOfWeek.Wednesday
            Debut = MaDate.AddDays(-2)
        Case DayOfWeek.Thursday
            Debut = MaDate.AddDays(-3)
        Case DayOfWeek.Friday
            Debut = MaDate.AddDays(-4)
        Case DayOfWeek.Saturday
            Debut = MaDate.AddDays(-5)
        Case DayOfWeek.Sunday
            Debut = MaDate.AddDays(-6)
    End Select
    Fin = Debut.AddDays(5)
    .SelectionRange = New SelectionRange(Debut, Fin)
End With
```

Notez que je pourrais utiliser les propriétés SelectionStart et SelectionEnd en lieu et place de SelectionRange.

HScrollBar & VScrollBar

Les barres de défilement sont maintenant très souvent directement intégrées aux contrôles qui en ont besoin, dès lors l'utilisation de ces deux contrôles est assez rare. On peut toutefois imaginer des cas ou il serait intéressant de s'en servir. Dans l'exemple suivant, un contrôle GroupBox contient un contrôle PictureBox. On donne à celui-ci la propriété AutoSize pour qu'il se dimensionne à la taille de l'image. Si celle-ci est supérieure à la taille du GroupBox, on ajoute des barres de défilement pour pouvoir visualiser l'image sans modifier la taille de la zone de groupe.

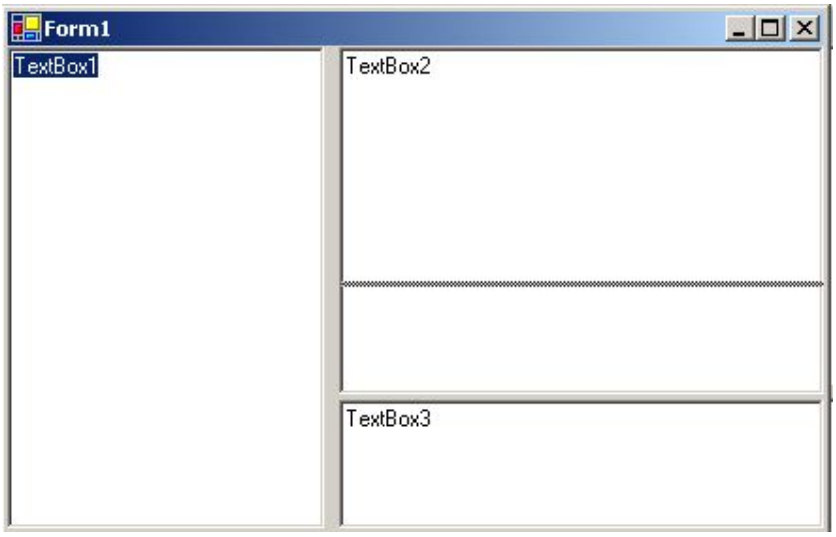
```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    Me.PictureBox1.SizeMode = PictureBoxSizeMode.AutoSize
    If Me.PictureBox1.Size.Height > Me.GroupBox1.Size.Height Then
        Dim MaScrollVert As New VScrollBar
        Me.GroupBox1.Controls.Add(MaScrollVert)
        Me.PictureBox1.SendToBack()
        With MaScrollVert
            .Dock = DockStyle.Left
            .Minimum = 0
            .Maximum = CInt(Me.PictureBox1.Size.Height)
            .Value = 0
            .SmallChange = CInt(Me.PictureBox1.Size.Height / 100)
            .LargeChange = CInt(Me.PictureBox1.Size.Height / 10)
        End With
        AddHandler MaScrollVert.Scroll, AddressOf VScrollBar_Scroll
    End If
    If Me.PictureBox1.Size.Width > Me.GroupBox1.Size.Width Then
        Dim MaScrollHor As New HScrollBar
        Me.GroupBox1.Controls.Add(MaScrollHor)
        Me.PictureBox1.SendToBack()
        With MaScrollHor
            .Dock = DockStyle.Bottom
            .Minimum = 0
            .Maximum = CInt(Me.PictureBox1.Size.Width)
            .Value = 0
            .SmallChange = CInt(Me.PictureBox1.Size.Width / 100)
            .LargeChange = CInt(Me.PictureBox1.Size.Width / 10)
        End With
        AddHandler MaScrollHor.Scroll, AddressOf HScrollBar_Scroll
    End If

End Sub

Private Sub VScrollBar_Scroll(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.ScrollEventArgs)
    Me.PictureBox1.Top = (-1) * e.NewValue
End Sub

Private Sub HScrollBar_Scroll(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.ScrollEventArgs)
    Me.PictureBox1.Left = (-1) * e.NewValue
End Sub
```

Splitter

Le contrôle splitter ne demande généralement aucun code. Il agit comme barre de redimensionnement. Pour vous en servir, prenons l'exemple suivant :

1) Mettez un Panel sur la feuille et mettez sa propriété Dock à 'Fill'
2) Ajoutez un TextBox au Panel, mettez MultiLine à 'True' et Dock à 'Left'
3) Ajoutez un Splitter au Panel, mettez Dock à 'Left'
4) Ajoutez un TextBox au Panel, mettez MultiLine à 'True' et Dock à 'Top'
5) Ajoutez un Splitter au Panel, mettez Dock à 'Top'
6) Ajoutez un TextBox au Panel, mettez MultiLine à 'True' et Dock à Fill

Vous avez maintenant trois zones de textes redimensionnables telles que :



DomainUpDown

On peut voir ce contrôle comme une zone de liste un peu particulière. Elle fonctionne à l'identique mais n'affiche qu'un élément. Le changement d'élément se fait par des flèches de navigation.

On l'utilise généralement dans les cas ou il ne peut y avoir qu'un élément sélectionné et quand il y a une notion de navigation induite par les éléments de la liste. Imaginons la gestion d'un petit emploi du temps. Mon contrôle va gérer les plages d'une heure contenues dans une journée. Le texte de chaque plage sera saisi/affiché par une zone de texte et manipulé par l'intermédiaire d'une 'StringCollection'.

```
Private Planning As System.Collections.Specialized.StringCollection
Private LastIndex As Int32

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim cmpt As Int32
    Planning = New Collections.Specialized.StringCollection
    For cmpt = 0 To 23
        Planning.Add("")
        Me.DomainUpDown1.Items.Add(cmpt & "h - " & cmpt + 1 & "h")
    Next
    Me.DomainUpDown1.SelectedIndex = 0
End Sub

Private Sub DomainUpDown1_SelectedItemChanged(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
DomainUpDown1.SelectedItemChanged
    If Me.DomainUpDown1.SelectedIndex = -1 Then Exit Sub
    Planning.Item(LastIndex) = Me.TextBox1.Text
    Me.TextBox1.Text = Planning.Item(Me.DomainUpDown1.SelectedIndex)
    LastIndex = Me.DomainUpDown1.SelectedIndex
End Sub
```

Certains développeurs utilisent ce type de contrôle comme une zone de liste pour gagner de la place. C'est très souvent une faute d'ergonomie. Les contrôles de type Up/Down doivent être réservés lorsque la notion de navigation existe dans les éléments. Notez que si la propriété 'ReadOnly' du contrôle est à 'False', il y a auto-complétion du texte saisi.

NumericUpDown

Il s'agit de l'équivalent du précédent mais pour des valeurs numériques. Ce contrôle peut être borné grâce à ses propriétés Minimum et Maximum, et la valeur se modifie par la propriété 'Increment'. Il ne fonctionne donc plus par une liste d'Item mais par manipulation numérique.

Autrement dit, un clic sur la flèche du haut ajoute 'Increment' à 'Value', un clic sur la flèche du bas retranche 'Increment' à 'Value'.

Un avantage certain de ce type de contrôle réside dans le fait qu'il est typé et donc qu'il simplifie le code de validation.

Notez qu'il permet aussi de travailler sur des hexadécimaux. Le code suivant permet de modifier la couleur de fond de la feuille avec trois contrôles NumericUpDown générant les valeurs RGB.

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    With nudR
        .Minimum = 0
        .Maximum = 255
        .Increment = 10
        .Value = 212
    End With
    With nudG
        .Minimum = 0
        .Maximum = 255
        .Increment = 10
        .Value = 208
    End With
    With nudB
        .Minimum = 0
        .Maximum = 255
        .Increment = 10
        .Value = 200
    End With
End Sub

Private Sub NumericUpDown_ValueChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles nudR.ValueChanged,
nudB.ValueChanged, nudG.ValueChanged
    Me.BackColor = Color.FromArgb(Decimal.ToInt32(nudR.Value),
Decimal.ToInt32(nudG.Value), Decimal.ToInt32(nudB.Value))
End Sub

Private Sub nudB_Validating(ByVal sender As Object, ByVal e As
System.ComponentModel.CancelEventArgs) Handles nudB.Validating,
nudR.Validating, nudG.Validating
    Me.BackColor = Color.FromArgb(Decimal.ToInt32(nudR.Value),
Decimal.ToInt32(nudG.Value), Decimal.ToInt32(nudB.Value))
End Sub
```

Il est possible de changer la valeur avec le clavier si la propriété 'ReadOnly' du contrôle vaut 'False'. Cependant, l'évènement ValueChanged ne sera déclenché que si l'utilisateur finit sa saisie par 'Enter'. C'est pour cela que j'intercepte aussi l'évènement Validating.

TrackBar

Assez proche des contrôles scrollBar, ce contrôle permet normalement la sélection d'une valeur numérique à l'aide d'un curseur.

Le code suivant permet le réglage de la transparence de la fenêtre à l'aide d'un TrackBar

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    With Me.TrackBar1
        .Minimum = 0
        .Maximum = 100
        .Value = 100
        .Dock = DockStyle.Bottom
        .LargeChange = 10
        .SmallChange = 1
    End With
End Sub

Private Sub TrackBar1_Scroll(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles TrackBar1.Scroll
    Me.Opacity = Me.TrackBar1.Value / 100
End Sub
```

ProgressBar

On utilise ce contrôle pour illustrer la progression d'une tâche. Ceci demande de pouvoir évaluer a priori la durée de la tâche, on l'utilise donc principalement avec des boucles For. Le principe est toujours le même. Une propriété Value se déplace de la valeur minimum vers la valeur maximum. Soit on modifie directement la propriété Value, soit on définit la propriété Step et on appelle la méthode PerformStep. On peut aussi utiliser la méthode Increment si la progression n'est pas régulière.

Cet exemple montre la progression du chargement d'un fichier texte dans un tableau.

```
Dim Lecture As New IO.StreamReader("d:\user\tutos\migration3\test.dat")
Dim Recup As String = Lecture.ReadToEnd
Lecture.Close()
Dim TabString() As String = Recup.Split(New [Char]() {Chr(13)})
Dim NbElem As Int32 = TabString(2).Split(New [Char]()
{Chr(9)}).GetLength(0)
Dim TabFin(TabString.GetLength(0), NbElem) As String
With Me.ProgressBar1
    .Visible = True
    .Minimum = 0
    .Maximum = TabString.GetLength(0)
    .Step = 1
End With
For cmpt1 As Int32 = 0 To TabString.GetLength(0) - 1
    For cmpt2 As Int32 = 0 To NbElem - 1
        TabFin(cmpt1, cmpt2) = CType(TabString(cmpt1).Split(New [Char]()
{Chr(9)}).GetValue(cmpt2), String)
    Next
    Me.ProgressBar1.PerformStep()
Next
```

Cette méthode trace une progression correcte mais m'oblige à faire une petite manipulation à seule fin de connaître le nombre d'éléments.

RichTextBox

Il s'agit d'un TextBox permettant la manipulation du format RichText. Celui-ci étant déjà assez complexe, il n'est pas question ici de montrer toutes les manipulations possibles. Cependant, l'exemple suivant va en illustrer quelques-unes.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    With Me.RichTextBox1
```

```
        .AppendText("De l'aide en VB.NET")
        .SelectAll()
        .SelectionFont = New Font("Arial", 14, FontStyle.Bold)
        .SelectionColor = Color.Blue
        .AppendText(vbCrLf)
        .AppendText(vbCrLf)
        .SelectionFont = New Font("Times new roman", 12)
        .AppendText("Nous pouvons en trouver sur :")
        .AppendText(vbCrLf)
        .SelectionBullet = True
        .SelectedText = "www.microsoft.com/france/vbasic/default.mspx" +
ControlChars.Cr
        .SelectedText = "www.vb.developpez.com" + ControlChars.Cr
        .SelectionBullet = False
        .SelectedText = ControlChars.NewLine
        .DetectUrls = True
    End With
    Dim MonImage As New Bitmap("d:\user\tutos\migration3\logo.bmp")
    Clipboard.SetDataObject(MonImage)
    If
Me.RichTextBox1.CanPaste(DataFormats.GetFormat(DataFormats.Bitmap)) Then
       Me.RichTextBox1.Paste(DataFormats.GetFormat(DataFormats.Bitmap))
    End If

End Sub

Private Sub RichTextBox1_LinkClicked(ByVal sender As Object, ByVal e As
System.Windows.Forms.LinkClickedEventArgs) Handles
RichTextBox1.LinkClicked

    If e.LinkText.EndsWith("com") Then
        System.Diagnostics.Process.Start("IExplore.exe",
"vb.developpez.com")
    Else
        System.Diagnostics.Process.Start(e.LinkText)
    End If

End Sub
```

Comme vous le voyez, je peux utiliser indifféremment AppendText ou SelectedText pour entrer du texte. S'il n'y a pas de texte sélectionné, les propriétés de type SelectionFont, SelectionBullet, etc… s'appliquent au texte qui suit jusqu'à qu'elles soient modifiées de nouveau.

Notez que le contrôle RichTextBox VB.NET n'expose plus de propriété OLEObjects, c'est pour cela que je passe par le presse-papiers pour insérer une image.

L'évènement LinkClicked est utilisé uniquement lorsque la propriété DetectUrls est vraie.

Le résultat final sera :

ToolTip

Ce contrôle est une nouvelle façon d'aborder les fonctionnalités communes des contrôles. Dans VB 6, les contrôles avaient chacun une propriété ToolTipText dans laquelle on pouvait mettre la chaîne qui devait apparaître dans le ToolTip. Celui-ci n'était pas manipulable ce qui fait qu'on se retrouvait souvent obligé de gérer un Label pour simuler le ToolTip.

ToolTip est un composant, il s'ajoute donc dans la barre des composants de la fenêtre. Dès lors, il ajoute une propriété « ToolTip sur *ToolTip1*(nom du composant) » ce qui permet d'associer le texte au contrôle en mode conception. Le composant ToolTip expose quand à lui les propriétés de fonctionnement suivantes :

Active	Obtient ou définit une valeur indiquant si l'info-bulle est actuellement active.
AutomaticDelay	Obtient ou définit le délai initial pour l'info-bulle.
AutoPopDelay	Obtient ou définit la durée d'affichage du ToolTip lorsque le pointeur de la souris s'immobilise dans un contrôle contenant le texte info-bulle spécifié.
InitialDelay	Obtient ou définit le temps écoulé avant l'apparition du ToolTip.
ReshowDelay	Obtient ou définit le délai qui doit s'écouler avant que s'affichent des fenêtres ToolTip qui se suivent lorsque le pointeur de la souris passe d'une zone de contrôle à une autre.
ShowAlways	Obtient ou définit une valeur indiquant si la fenêtre ToolTip apparaît même lorsque son contrôle parent n'est pas actif.

On manipule les textes de l'info-bulle à l'aide des méthodes SetToolTip et GetToolTip.

Par exemple, la ligne suivante attribue comme texte à l'info bulle d'une zone de liste le texte de l'élément sélectionné :

```
Me.ToolTip1.SetToolTip(Me.ListBox1, CType(Me.ListBox1.SelectedItem,
String))
```

ErrorProvider

Petit nouveau dans la famille des contrôles Visual Basic, le contrôle ErrorProvider fournit une indication visuelle d'erreur pour la validation des entrées de l'utilisateur. Il fonctionne sur le même principe que le contrôle ToolTip, c'est-à-dire qu'on doit l'associer aux contrôles en erreur. On place généralement son code de gestion dans l'évènement Validating du contrôle cible. L'exemple suivant notifie une erreur si la valeur saisie dans une zone de texte n'est pas numérique.

```
Private Sub TextBox1_Validating(ByVal sender As Object, ByVal e As
System.ComponentModel.CancelEventArgs) Handles TextBox1.Validating
    If Not IsNumeric(Me.TextBox1.Text) Then
        Me.ErrorProvider1.SetError(Me.TextBox1, "L'entrée doit être
numérique")
        e.Cancel = True
    Else
        Me.ErrorProvider1.SetError(Me.TextBox1, "")
    End If
End Sub
```

Vous pouvez aussi modifier l'apparence du contrôle à l'aide des propriétés BlinkStyle et BlinkRate.

StatusBar

C'est la barre d'état. Vous pouvez l'utiliser comme une simple zone de message ou gérer des zones multiples, par l'intermédiaire de sa collection Panels. Elle est très simple à utiliser. Dans l'exemple suivant, la barre affiche l'état des touches CapsLock et NumLock, la date du jour et l'heure.

Notez que j'utilise une API Win32, de la même façon qu'en VB6 pour aller lire l'état des touches.

```
Private Declare Function GetKeyboardState Lib "user32" (ByRef pbKeyState
As Byte) As Integer
Const VK_NUMLOCK As Short = &H90S
Const VK_CAPSLOCK As Short = &H14S

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    With Me.StatusBar1
        .Panels.Add("CapsLock")
        .Panels.Add("NumLock")
        .Panels.Add(Now.ToLongDateString)
        .Panels.Add(Now.ToLongTimeString)
        .ShowPanels = True
    End With
    Me.Timer1.Interval = 1000
    Me.Timer1.Start()
End Sub

Private Sub Timer1_Tick(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Timer1.Tick
    Dim Touches(255) As Byte
    GetKeyboardState(Touches(0))

    If CBool(Touches(VK_CAPSLOCK)) Then
        Me.StatusBar1.Panels(0).Text = "CapsLock"
    Else
        Me.StatusBar1.Panels(0).Text = ""
    End If
    If CBool(Touches(VK_NUMLOCK)) Then
        Me.StatusBar1.Panels(1).Text = "NumLock"
```

```
    Else
        Me.StatusBar1.Panels(1).Text = ""
    End If
    Me.StatusBar1.Panels(3).Text = Now.ToLongTimeString
End Sub
```

Boites de dialogue standards

Avec VB 6, un seul composant gérait les cinq boites de dialogue standard, à savoir Ouvrir, Enregistrer, Imprimer, Couleur et Police. Il existe maintenant un composant pour chacune, plus deux nouvelles, *Aperçu avant impression* et *Parcourir*.

Comme il s'agit maintenant de composants séparés le point d'entrée est toujours la méthode ShowDialog. Pour la plupart de ces boites, elles ne renvoient qu'un objet simple c'est-à-dire la sélection de l'utilisateur, ce qui les rendent d'un usage aisé.

Les boites de dialogue d'impression sont un peu plus complexe à utiliser, nous les verrons donc lorsque nous traiterons de l'impression un peu plus loin dans ce cours.

Nous verrons un exemple dans l'étude du contrôle suivant.

ToolBar

Le contrôle affiche une barre d'outils. Généralement cette barre donne un accès rapide à des éléments de menu mais ce n'est pas obligatoire. Une barre d'outils gère quatre types de bouton :

- Les boutons standards (Style PushButton) répètent la même action à chaque clic.
- Les boutons à bascule (Style ToggleButton) ont deux états (relevé/enfoncé).
- Les boutons déroulant (Style DropDownButton) affichent un menu
- Les séparateurs (Style Separator) séparent les groupes de boutons

Le contrôle ToolBar gère aussi les info-bulles sans utiliser le contrôle ToolTip.

Dans l'exemple suivant qui va nous servir jusqu'à la fin de ce chapitre, je vais travailler sur un contrôle RichTextBox comme éditeur de texte. Je vais ajouter à ma feuille une barre d'outil. La définition par le code de celle-ci sera :

```
Me.ToolBar1 = New System.Windows.Forms.ToolBar
Me.tbbOpen = New System.Windows.Forms.ToolBarButton
Me.tbbSave = New System.Windows.Forms.ToolBarButton
Me.tbbSep1 = New System.Windows.Forms.ToolBarButton
Me.tbbFont = New System.Windows.Forms.ToolBarButton
Me.tbbColor = New System.Windows.Forms.ToolBarButton
Me.tbbSep2 = New System.Windows.Forms.ToolBarButton
     Me.tbbLock = New System.Windows.Forms.ToolBarButton
'
'ToolBar1
'
Me.ToolBar1.Buttons.AddRange(New System.Windows.Forms.ToolBarButton()
{Me.tbbOpen, Me.tbbSave, Me.tbbSep1, Me.tbbFont, Me.tbbColor, Me.tbbSep2,
Me.tbbLock})
Me.ToolBar1.DropDownArrows = True
Me.ToolBar1.ImageList = Me.ImageList1
Me.ToolBar1.Location = New System.Drawing.Point(0, 0)
Me.ToolBar1.Name = "ToolBar1"
Me.ToolBar1.ShowToolTips = True
Me.ToolBar1.Size = New System.Drawing.Size(624, 44)
Me.ToolBar1.TabIndex = 6
'
'tbbOpen
'
Me.tbbOpen.ImageIndex = 0
Me.tbbOpen.ToolTipText = "Ouvrir"
```

```
'
'tbbSave
'
Me.tbbSave.ImageIndex = 1
Me.tbbSave.ToolTipText = "Enregistrer"
'
'tbbSep1
'
Me.tbbSep1.Style = System.Windows.Forms.ToolBarButtonStyle.Separator
'
'tbbFont
'
Me.tbbFont.ImageIndex = 2
Me.tbbFont.ToolTipText = "Police"
'
'tbbColor
'
Me.tbbColor.ImageIndex = 3
Me.tbbColor.ToolTipText = "Couleur"
'
'tbbSep2
'
Me.tbbSep2.Style = System.Windows.Forms.ToolBarButtonStyle.Separator
'
'tbbLock
'
Me.tbbLock.ImageIndex = 4
Me.tbbLock.Style = System.Windows.Forms.ToolBarButtonStyle.ToggleButton
Me.tbbLock.ToolTipText = "Déverrouiller"
```

Ma feuille contient aussi les composants suivant : ImageList1, OpenFileDialog1, SaveFileDialog1, FontDialog1, ColorDialog1

Mon code sera donc le suivant :

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Me.tbbFont.Enabled = False
    Me.tbbColor.Enabled = False
    Me.RichTextBox1.Enabled = False
End Sub

Private Sub ToolBar1_ButtonClick(ByVal sender As System.Object, ByVal e
As System.Windows.Forms.ToolBarButtonClickEventArgs) Handles
ToolBar1.ButtonClick
    Select Case ToolBar1.Buttons.IndexOf(e.Button)
        Case 0 'ouvrir
            With Me.OpenFileDialog1
                .CheckFileExists = True
                .CheckPathExists = True
                .Filter = "Fichiers texte (*.txt)|*.txt|Fichiers RichText
(*.rtf)|*.rtf"
                .FilterIndex = 2
                .Multiselect = False
            End With
            If OpenFileDialog1.ShowDialog() = DialogResult.OK Then
                Me.RichTextBox1.LoadFile(Me.OpenFileDialog1.FileName)
            End If
```

```
        Case 1 'Enregistrer
            With Me.SaveFileDialog1
                .AddExtension = True
                .DefaultExt = "rtf"
                .CheckPathExists = True
                .CheckFileExists = True
                .ValidateNames = True
            End With
            If Me.SaveFileDialog1.ShowDialog = DialogResult.OK Then
                Me.RichTextBox1.SaveFile(Me.SaveFileDialog1.FileName)
            End If

        Case 3 'Police
            If Me.RichTextBox1.SelectionLength > 0 Then
                If Me.FontDialog1.ShowDialog = DialogResult.OK Then
Me.RichTextBox1.SelectionFont = Me.FontDialog1.Font
            End If

        Case 4 'Couleur
            If Me.RichTextBox1.SelectionLength > 0 Then
                If Me.ColorDialog1.ShowDialog = DialogResult.OK Then
Me.RichTextBox1.SelectionColor = Me.ColorDialog1.Color
            End If

        Case 6 'verrouiller
            If e.Button.Pushed = True Then
                If Me.RichTextBox1.TextLength > 0 Then
                    Me.ToolBar1.Buttons(3).Enabled = True
                    Me.ToolBar1.Buttons(4).Enabled = True
                    Me.RichTextBox1.Enabled = True
                    e.Button.ImageIndex = 5
                Else
                    Me.ToolBar1.Buttons(3).Enabled = False
                    Me.ToolBar1.Buttons(4).Enabled = False
                    Me.RichTextBox1.Enabled = False
                    e.Button.ImageIndex = 4
                    e.Button.Pushed = False
                End If
            Else
                e.Button.ImageIndex = 4
                Me.RichTextBox1.Enabled = False
            End If

    End Select
End Sub
```

Ceci nous conduit tout naturellement à parler des menus. Il s'agit finalement de la même approche que pour la barre d'outils

Menu

Dans VB 6, il faut forcément créer un menu dans l'IDE pour pouvoir le manipuler par le code. Dans le monde DotNet, les menus peuvent être créés et manipulés par le code. Qui plus est, on peut faire des menus à présentation plus évoluée sans devoir appeler l'API Windows. Il existe deux contrôles gérant des menus, MainMenu et ContextMenu.

MainMenu

Toujours dans notre exemple, nous allons ajouter un contrôle MainMenu à la feuille pour créer les mêmes menus que ceux gérés implicitement dans la barre d'outils. J'ajoute des raccourcis à mes menus (propriété Shortcut). La définition par le code est :

```
Me.MainMenu1.MenuItems.AddRange(New System.Windows.Forms.MenuItem()
{Me.mnuFichier, Me.mnuMForme})
'
'mnuFichier
Me.mnuFichier.Index = 0
Me.mnuFichier.MenuItems.AddRange(New System.Windows.Forms.MenuItem()
{Me.mnuOuvrir, Me.mnuSave})
Me.mnuFichier.Shortcut = System.Windows.Forms.Shortcut.CtrlF
Me.mnuFichier.Text = "Fichier"
'
'mnuOuvrir
Me.mnuOuvrir.Index = 0
Me.mnuOuvrir.Shortcut = System.Windows.Forms.Shortcut.CtrlO
Me.mnuOuvrir.Text = "Ouvrir"
'
'mnuSave
Me.mnuSave.Index = 1
Me.mnuSave.Shortcut = System.Windows.Forms.Shortcut.CtrlE
Me.mnuSave.Text = "Enregistrer"
'mnuMForme
Me.mnuMForme.Enabled = False
Me.mnuMForme.Index = 1
Me.mnuMForme.MenuItems.AddRange(New System.Windows.Forms.MenuItem()
{Me.mnuFont, Me.mnuColor, Me.mnuSep1, Me.mnuLock})
Me.mnuMForme.Shortcut = System.Windows.Forms.Shortcut.CtrlM
Me.mnuMForme.Text = "Mise en forme"
'
'mnuFont
Me.mnuFont.Enabled = False
Me.mnuFont.Index = 0
Me.mnuFont.Shortcut = System.Windows.Forms.Shortcut.CtrlP
Me.mnuFont.Text = "Police"
'
'mnuColor
Me.mnuColor.Enabled = False
Me.mnuColor.Index = 1
Me.mnuColor.Shortcut = System.Windows.Forms.Shortcut.CtrlL
Me.mnuColor.Text = "Couleur"
'
'mnuSep1
Me.mnuSep1.Index = 2
Me.mnuSep1.Text = "-"
'
```

```
'mnuLock
Me.mnuLock.Index = 3
Me.mnuLock.Shortcut = System.Windows.Forms.Shortcut.CtrlD
Me.mnuLock.Text = "Déverrouiller"
```

La gestion du code des menus se fait alors sur les évènements clic des éléments du menu. Dans notre cas nous aurons :

```
Private Sub mnuOuvrir_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuOuvrir.Click
    With Me.OpenFileDialog1
        .CheckFileExists = True
        .CheckPathExists = True
        .Filter = "Fichiers texte (*.txt)|*.txt|Fichiers RichText
(*.rtf)|*.rtf"
        .FilterIndex = 2
        .Multiselect = False
    End With
    If OpenFileDialog1.ShowDialog() = DialogResult.OK Then
        Me.RichTextBox1.LoadFile(Me.OpenFileDialog1.FileName)
        Me.MainMenu1.MenuItems(1).Enabled = True
    End If
End Sub

Private Sub mnuSave_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuSave.Click
    With Me.SaveFileDialog1
        .AddExtension = True
        .DefaultExt = "rtf"
        .CheckPathExists = True
        .CheckFileExists = True
        .ValidateNames = True
    End With
    If Me.SaveFileDialog1.ShowDialog = DialogResult.OK Then
        Me.RichTextBox1.SaveFile(Me.SaveFileDialog1.FileName)
    End If
End Sub

Private Sub mnuLock_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuLock.Click
    With mnuLock
        If .Checked = False Then
            If Me.RichTextBox1.TextLength > 0 Then
                Me.MainMenu1.MenuItems(1).MenuItems(0).Enabled = True
                Me.MainMenu1.MenuItems(1).MenuItems(1).Enabled = True
            Else
                Me.MainMenu1.MenuItems(1).MenuItems(0).Enabled = False
                Me.MainMenu1.MenuItems(1).MenuItems(1).Enabled = False
            End If
            Me.RichTextBox1.Enabled = True
            .Text = "Verrouiller"
            .Checked = True
        Else
            Me.RichTextBox1.Enabled = False
            .Text = "Déverrouiller"
            .Checked = False
        End If
```

```
    End With
End Sub

Private Sub mnuFont_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuFont.Click
    If Me.RichTextBox1.SelectionLength > 0 Then
        If Me.FontDialog1.ShowDialog = DialogResult.OK Then
Me.RichTextBox1.SelectionFont = Me.FontDialog1.Font
    End If
End Sub

Private Sub mnuColor_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuColor.Click
    If Me.RichTextBox1.SelectionLength > 0 Then
        If Me.ColorDialog1.ShowDialog = DialogResult.OK Then
Me.RichTextBox1.SelectionColor = Me.ColorDialog1.Color
    End If
End Sub
```

A peu de chose près il s'agit des mêmes codes que pour les boutons de la barre d'outils. Pour éviter de doubler le code, on appelle alors les évènements de menu dans le code de gestion de la barre d'outils. La procédure ressemble alors à :

```
Private Sub ToolBar1_ButtonClick(ByVal sender As System.Object, ByVal e
As System.Windows.Forms.ToolBarButtonClickEventArgs) Handles
ToolBar1.ButtonClick
    Select Case ToolBar1.Buttons.IndexOf(e.Button)
        Case 0 'ouvrir
            Me.mnuOuvrir_Click(sender, e)

        Case 1 'Enregistrer
            Me.mnuSave_Click(sender, e)

        Case 3 'Police
            Me.mnuFont_Click(sender, e)

        Case 4 'Couleur
            Me.mnuColor_Click(sender, e)

        Case 6 'verrouiller
            If e.Button.Pushed = True Then
                If Me.RichTextBox1.TextLength > 0 Then
                    Me.ToolBar1.Buttons(3).Enabled = True
                    Me.ToolBar1.Buttons(4).Enabled = True
                    e.Button.ImageIndex = 5
                Else
                    Me.ToolBar1.Buttons(3).Enabled = False
                    Me.ToolBar1.Buttons(4).Enabled = False
                    e.Button.ImageIndex = 4
                    e.Button.Pushed = False
                End If
            Else
                e.Button.ImageIndex = 4
            End If
            Me.mnuLock_Click(sender, e)
    End Select
End Sub
```

ContextMenu

Ce composant gère les menus contextuels. Comme le composant ToolTip, il est partagé c'est-à-dire qu'il permet de gérer les menus contextuels de plusieurs contrôles. Un menu contextuel est appelé par un click droit sur le contrôle. Le contrôle ContextMenu n'expose pas de collection MenuItems en mode conception, on le remplit donc toujours par le code. Généralement, il n'expose que des menus tirés du menu principal, mais ce n'est pas une obligation comme nous allons le voir.

Je vais ajouter un menu contextuel. Celui-ci exposera les menus « couleur » et « police » du menu principal, mais exposera aussi des menus « copier » et « coller ».

```
Private Sub RichTextBox1_MouseDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles RichTextBox1.MouseDown
    If e.Button = MouseButtons.Right Then
        If Me.RichTextBox1.SelectionLength > 0 Then
            Me.RichTextBox1.ContextMenu = Me.ContextMenu1
            If Me.ContextMenu1.MenuItems.Count = 0 Then
                With Me.ContextMenu1
                    'ajout menus existant
                    .MenuItems.Add(Me.mnuColor)
                    .MenuItems.Add(Me.mnuFont)
                    'création dynamique
                    .MenuItems.Add("Copier", New
System.EventHandler(AddressOf Me.Copier_OnClick))
                    .MenuItems.Add("Coller", New
System.EventHandler(AddressOf Me.Coller_OnClick))
                End With
            End If
        End If
    End If
End Sub

Private Sub Copier_OnClick(ByVal sender As System.Object, ByVal e As
System.EventArgs)
    Clipboard.SetDataObject(Me.RichTextBox1.SelectedText)
End Sub

Private Sub Coller_OnClick(ByVal sender As System.Object, ByVal e As
System.EventArgs)

    If Me.RichTextBox1.CanPaste(DataFormats.GetFormat(DataFormats.Text))
Then
        Me.RichTextBox1.Paste(DataFormats.GetFormat(DataFormats.Text))
    End If
End Sub
```

Application MDI

Pour finir avant de se lancer dans les joies des contrôles personnalisés, nous allons modifier notre exemple pour voir la gestion des documents multiples, autrement dit les formulaires MDI.

Les applications MDI se gèrent toujours de la même façon. Un formulaire hôte appelé « formulaire parent » contient un ou plusieurs formulaires de même type.

Le gros du travail a déjà été fait dans notre exemple précédent. J'ajoute un menu « Fermer » dans mon menu fichier et un menu « Fenêtre » au premier niveau de mon contrôle MainMenu1. Ce menu « Fenêtre » doit avoir Vrai à sa propriété MDIList. Enfin, je supprime le menu « Mise en forme ».

Je mets à Vrai la propriété '*IsMDIContainer*' de mon formulaire.

J'ajoute un nouveau formulaire à ma solution. Je déplace mon RichtextBox dans le nouveau formulaire. J'ajoute un contrôle MainMenu à ce nouveau formulaire puis j'y place mon menu mise en forme (avec un couper/coller). Je déplace aussi les boites de dialogue *Police* et *Couleur* ainsi que le composant ContextMenu1.

Le code de mon formulaire parent devient alors :

```
Private Sub mnuOuvrir_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuOuvrir.Click
    With Me.OpenFileDialog1
        .CheckFileExists = True
        .CheckPathExists = True
        .Filter = "Fichiers texte (*.txt)|*.txt|Fichiers RichText
(*.rtf)|*.rtf"
        .FilterIndex = 2
        .Multiselect = False
    End With
    If OpenFileDialog1.ShowDialog() = DialogResult.OK Then
        Dim NouvFeuilleMDI As New Form2(Me.OpenFileDialog1.FileName)
        NouvFeuilleMDI.MdiParent = Me
        NouvFeuilleMDI.Show()
    End If
    End Sub

Private Sub mnuSave_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuSave.Click
    If Me.MdiChildren.Length = 0 Then Exit Sub
    With Me.SaveFileDialog1
        .AddExtension = True
        .DefaultExt = "rtf"
        .CheckPathExists = True
        .CheckFileExists = True
        .ValidateNames = True
    End With
    If Me.SaveFileDialog1.ShowDialog = DialogResult.OK Then
        Dim FeuilleActive As Form2
        FeuilleActive = CType(Me.ActiveMdiChild, Form2)
        FeuilleActive.RichTextBox1.SaveFile(Me.SaveFileDialog1.FileName)
    End If
End Sub
Private Sub mnuClose_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuClose.Click
    If Me.MdiChildren.Length > 0 Then
        Me.ActiveMdiChild.Close()
    End If
End Sub
```

Notez que pour passer le nom de fichier choisi dans la feuille parent, j'ai surchargé la méthode New de la feuille enfant « Form2 » tel que :

```
Public Sub New(ByVal NomFichier As String)
    MyBase.New()
    InitializeComponent()
    Me.RichTextBox1.LoadFile(NomFichier)
    Me.MainMenu1.MenuItems(0).Enabled = True
    Me.Text = New IO.FileInfo(NomFichier).Name
End Sub
```

Ceci n'est qu'une ébauche bien sur, mais la manipulation des applications MDI est la même qu'en VB6 si ce n'est qu'on utilise ActiveMdiChild au lieu d'ActiveForm.

Contrôles personnalisés

Avec VB6, il n'existait pas beaucoup de solutions pour personnaliser les contrôles, il fallait passer par un contrôle utilisateur (UserControl) ou travailler sur le code évènementiel du contrôle dans le formulaire.

Maintenant, l'éventail de possibilités est beaucoup plus vaste, du fait de l'héritage. Ce sujet est immensément vaste, aussi n'allons nous voir que quelques approches différentes, illustrées d'exemple.

Dérivation de contrôle

Cette approche est intéressante puisqu'elle couvre un grand nombre de possibilités avec un travail relativement réduit. Elle peut cependant être limitée pour des personnalisations importantes.

Ce type de dérivation peut être implémenté dans des formulaires, dans des composants du projet ou dans une bibliothèque de contrôles, le choix dépendant de la mise à disposition du contrôle. Cela signifie que si je dérive un contrôle sans l'espoir de le réutiliser, je peux le faire directement dans ma feuille, sinon il y a plutôt intérêt à utiliser un composant.

Ne cherchez surtout pas à faire ces dérivations dans une bibliothèque de classe car les contrôles ont une interface graphique, utilisent l'espace de nom System.Windows.Forms et doivent être créés dans une entité permettant d'importer cet espace de nom.

Dans mon cas, je vais créer une nouvelle solution dans laquelle j'ajoute deux composants, un fichier de classe ClsDerivation et un formulaire WindowsForms 'frmDerivation'.

Lançons-nous dans quelques exemples concrets.

TextBox numérique

Dans ce premier exemple, je vais dériver une zone de texte pour la transformer en une zone permettant uniquement la saisie d'un entier 32 bit. C'est un cas simple car il suffit de redéfinir la méthode protégée OnKeyPress (celle qui est appelée par l'évènement KeyPress).

Le code est :

```
Public Class NumBox
    Inherits Windows.Forms.TextBox
    Friend WithEvents TextBox1 As System.Windows.Forms.TextBox

    Public Sub New()
        MyBase.New()
    End Sub

    Protected Overrides Sub OnKeyPress(ByVal e As
System.Windows.Forms.KeyPressEventArgs)
        Dim Test As String
        If Not Char.IsControl(e.KeyChar) Then
            If Char.IsDigit(e.KeyChar) Then
                If MyClass.SelectionLength > 0 Then
                    Test = MyClass.Text.Remove(MyClass.SelectionStart,
MyClass.SelectionLength)
                    Test = Test.Insert(MyClass.SelectionStart,
e.KeyChar.ToString)
                Else
                    Test = MyClass.Text.Insert(MyClass.SelectionStart,
e.KeyChar.ToString)
                End If
                Try
```

```
                    Dim intTest As Int32 = Int32.Parse(Test)
                    e.Handled = False
                Catch ex As Exception
                    e.Handled = True
                End Try
            Else
                e.Handled = True
            End If
        End If
    End Sub

End Class
```

Rappelons-nous que *MyClass* représente ma classe « NumBox » et que *MyBase* désigne la classe dont j'hérite, en l'occurrence « TextBox ».

DateTimePicker & la valeur NULL

Le contrôle DateTimePicker est très utile pour restreindre les saisies, mais en l'état on ne peut le lier avec une base de données que si on a la certitude que le champ ne contient pas de valeur NULL. La solution serait de modifier son utilisation en dérivant le contrôle et en redéfinissant la propriété Value.

Dans ce cas, nous allons devoir masquer la propriété Value du contrôle d'origine puisque nous allons modifier le type de 'Value'.

```
Public Class NullDateTimePicker
    Inherits Windows.Forms.DateTimePicker
    Private m_Value As Object
    Private m_TypeSaveFormat As DateTimePickerFormat
    Private m_strSaveFormat As String

    Public Sub New()
        MyBase.New()
        m_TypeSaveFormat = MyClass.Format
    End Sub

    Public Shadows Property value() As Object
        Get
            Return m_Value
        End Get
        Set(ByVal Value As Object)
            If Convert.IsDBNull(Value) Then
                If m_strSaveFormat <> " " Then
                    m_TypeSaveFormat = MyClass.Format
                    If MyClass.Format = DateTimePickerFormat.Custom Then
m_strSaveFormat = MyClass.CustomFormat
                End If
                m_Value = DBNull.Value
                MyClass.value = MyBase.MinDate
                MyClass.Format = DateTimePickerFormat.Custom
                MyClass.CustomFormat = " "
            Else
                m_Value = Value
                MyClass.Format = Me.m_TypeSaveFormat
                If MyClass.Format = DateTimePickerFormat.Custom Then
MyClass.CustomFormat = Me.m_strSaveFormat
            End If
        End Set
    End Property
```

```
End Class
```

ComboBox personnalisé

Nous allons dériver le contrôle ComboBox pour qu'il intègre les modifications données précédemment en exemple, c'est-à-dire l'ajustement de la taille de la liste, la fonctionnalité Ajout/Suppression et l'auto-complétion.

```
Public Class PersoCombo
    Inherits System.Windows.Forms.ComboBox


    Public Sub New()
        MyBase.New()
        MyClass.DrawMode = DrawMode.OwnerDrawVariable
    End Sub

    Protected Overrides Sub RefreshItem(ByVal index As Integer)
        MyBase.RefreshItem(index)
    End Sub

    Protected Overrides Sub SetItemsCore(ByVal items As
System.Collections.IList)
        MyBase.SetItemsCore(items)
    End Sub

    Protected Overrides Sub OnMeasureItem(ByVal e As
System.Windows.Forms.MeasureItemEventArgs)

        Dim stringSize As SizeF =
e.Graphics.MeasureString(CType(MyClass.Items(e.Index), String),
MyClass.Font)
        e.ItemWidth = CInt(stringSize.Width)
        e.ItemHeight = CInt(stringSize.Height)
        If e.ItemWidth > MyClass.DropDownWidth Then MyClass.DropDownWidth
= e.ItemWidth
        MyBase.OnMeasureItem(e)

    End Sub

    Protected Overrides Sub OnDrawItem(ByVal e As
System.Windows.Forms.DrawItemEventArgs)

        MyBase.OnDrawItem(e)
        Dim mSF As New StringFormat
        mSF.Alignment = StringAlignment.Near
        mSF.LineAlignment = StringAlignment.Center
        e.Graphics.DrawString(CType(MyClass.Items(e.Index), String),
Me.Font, New SolidBrush(Me.ForeColor), e.Bounds.Left, e.Bounds.Top +
CType(e.Bounds.Height / 2, Integer), mSF)

    End Sub

    Protected Overrides Sub OnKeyDown(ByVal e As
System.Windows.Forms.KeyEventArgs)

        If e.KeyCode = Keys.Delete Then
```

```
            If MyBase.Items.Contains(MyClass.Text) Then
                MyBase.Items.Remove(MyClass.Text)
            End If
        ElseIf e.KeyCode = Keys.Enter AndAlso Not
MyBase.Items.Contains(MyClass.Text) Then
            MyBase.Items.Add(MyClass.Text)
        End If

    End Sub

    Protected Overrides Sub OnTextChanged(ByVal e As System.EventArgs)

        If MyBase.FindString(MyClass.Text) > -1 Then
            Dim Pos As Int32 = MyClass.Text.Length
            MyClass.SelectedIndex = MyBase.FindString(MyClass.Text)
            MyClass.SelectionStart = Pos
            MyClass.SelectionLength = MyClass.Text.Length - Pos
        End If

    End Sub
End Class
```

Pour les fonctionnalités d'ajout, de suppression et d'auto-complétion, rien de bien spécial. Par contre, vous voyez que, pour ajuster la taille de la liste, je modifie les méthodes graphiques du contrôle. Tout ceci pour démontrer que l'on peut aller assez loin en dérivant des contrôles existants.

Néanmoins, les contrôles créés ont une utilité limitée et, donc, sont peu distribuables. En effet, une zone de saisie numérique exposant une propriété PasswordChar n'est pas très fonctionnelle. De même mon DateTimePicker joue sur un artifice pour gérer les valeurs NULL.

Enfin pour ma ComboBox, je ne peux pas la contrôler finement par héritage puisque, par exemple, les actions sur la collection Items ne sont pas directement modifiables.

Dans un certain nombre de cas, la dérivation simple ne suffit pas, il faut alors créer son contrôle. Cela permet aussi d'obtenir des contrôles qu'on ne pourrait avoir par dérivation.

Contrôle utilisateur (Control & UserControl)

Fondamentalement, ce que nous allons voir ici est semblable à ce que nous avons vu précédemment. Toute la question, lors de la création d'un contrôle personnalisé, est de savoir quelle va être la classe de base. Dans ce que nous venons de voir, j'ai utilisé comme classe de base des contrôles existants, implémentant une interface riche. Dès lors, si nous souhaitons exposer un contrôle que nous maîtrisons mieux, nous devons hériter d'une classe moins riche. La contrepartie est évidente, il y a beaucoup plus de travail à fournir. *Control* est la classe de base de tous les contrôles. Si vous n'héritez que d'elle, il faudra écrire l'ensemble du code de dessin du contrôle. C'est une approche qui peut être intéressante pour créer des contrôles très spécifiques. La hiérarchie des classes est :

```
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ScrollableControl
          System.Windows.Forms.ContainerControl
            System.Windows.Forms.UserControl
```

Plus je descends dans la hiérarchie, plus ma classe de base gère de fonctionnalités. Dans un sens cela veut dire moins de code à écrire ; néanmoins, ce n'est pas si simple. En effet, plus j'hérite de fonctionnalités, plus il y a de chances que j'hérite d'un fonctionnement qui ne me convient pas. Il peut y avoir alors un travail non négligeable pour modifier ce fonctionnement ou pour masquer certains membres.

UserControl permet de créer des contrôles en partant de contrôles existants qu'on place dans le concepteur. Généralement, on déclare les contrôles Private afin de n'exposer que la partie que l'on souhaite. Par contre, le concepteur expose ses membres. Libre à nous d'en masquer certain si nous le souhaitons.

Cette approche est très différente de la méthode VB6. Pour bien voir le principe, nous allons écrire en VB.NET un contrôle écrit en VB6 par Fred Just. Vous en trouverez le code source, ainsi que celui d'autres composants qu'il a écrit, à : http://fred.just.free.fr/francais/index.html

Création de MaxBox VB.NET

Dans le concept, il s'agit d'un Frame VB6 qui possède deux tailles, agrandie et rétrécie, le basculement entre les deux étant géré par un bouton de commande.

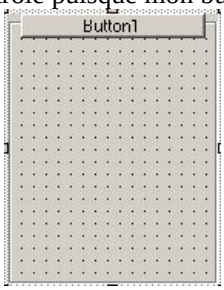
Je vais créer une nouvelle solution Windows Forms appelée « Test » dans mon IDE.

Dans le menu Projet, je choisis *'Ajouter un contrôle utilisateur'* que je nomme MaxBox.

L'IDE m'affiche alors le concepteur qui ressemble à un formulaire sans bordure ni barre de titre.

J'ajoute un contrôle Bouton à mon concepteur. Je place mon bouton en haut du concepteur. Je règle sa propriété Anchor sur (top, left, right). Dans notre cas, je ne vais pas ajouter de contrôle GroupBox. En effet un contrôle GroupBox n'est rien d'autre qu'un conteneur de contrôle comme mon UserControl (la base dont j'hérite). Si j'ajoute un contrôle GroupBox en le mettant par exemple en Fill de mon UserControl, je vais me retrouver à détourner toutes les propriétés d'apparence de mon UserControl vers le GroupBox.

De plus, les contrôles ajoutés par l'utilisateur devraient être redirigés vers le GroupBox ce qui est assez complexe à gérer. Toutefois pour garder la même apparence, je vais devoir tracer une bordure sur mon contrôle puisque mon but est d'obtenir un contrôle similaire à celui de Fred Just soit :



Au début, comme je n'ai ajouté qu'un bouton, le code généré par le concepteur est :

```
Public Class MaxBox
    Inherits System.Windows.Forms.UserControl

#Region " Code généré par le Concepteur Windows Form "

    Public Sub New()
        MyBase.New()

        'Cet appel est requis par le Concepteur Windows Form.
        InitializeComponent()

        'Ajoutez une initialisation quelconque après l'appel
InitializeComponent()

    End Sub
     'La méthode substituée Dispose du UserControl pour nettoyer la
liste des composants.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Requis par le Concepteur Windows Form
    Private components As System.ComponentModel.IContainer

    'REMARQUE : la procédure suivante est requise par le Concepteur
Windows Form
    'Elle peut être modifiée en utilisant le Concepteur Windows Form.
    'Ne la modifiez pas en utilisant l'éditeur de code.
    Friend WithEvents Button1 As System.Windows.Forms.Button
    <System.Diagnostics.DebuggerStepThrough()> Private Sub
InitializeComponent()
        Me.Button1 = New System.Windows.Forms.Button
        Me.SuspendLayout()
        '
        'Button1
        '
        Me.Button1.Anchor = CType(((System.Windows.Forms.AnchorStyles.Top
Or System.Windows.Forms.AnchorStyles.Left) _
                    Or System.Windows.Forms.AnchorStyles.Right),
System.Windows.Forms.AnchorStyles)
        Me.Button1.Location = New System.Drawing.Point(8, 0)
        Me.Button1.Name = "Button1"
        Me.Button1.Size = New System.Drawing.Size(120, 16)
        Me.Button1.TabIndex = 0
        Me.Button1.Text = "Button1"
        '
        'MaxBox
        '
        Me.Controls.Add(Me.Button1)
        Me.Name = "MaxBox"
```

```
        Me.Size = New System.Drawing.Size(136, 176)
        Me.ResumeLayout(False)

    End Sub

#End Region

End Class
```

Si je générais ma solution à cet instant, j'aurais un contrôle qui ne possèderais que les membres exposés par le concepteur, c'est-à-dire les membres de la classe UserControl. En effet, le bouton étant de portée privée, il n'expose pas de propriétés accessibles.

Masquage de membres hérités

Dès à présent, je pourrais souhaiter que des propriétés exposées par la classe de base, dans notre cas UserControl ne le soit plus. Une technique consiste à marque le membre d'un attribut afin qu'il ne soit pas exposé dans la fenêtre des propriétés et, d'un autre attribut pour le masquer dans Intellisense. Par exemple, si je désire faire disparaître la propriété AutoScroll du concepteur je peux déclarer :

```
<Browsable(False), EditorBrowsable(EditorBrowsableState.Never)> Overrides
Property AutoScroll() As Boolean
    Get
        Return False
    End Get
    Set(ByVal Value As Boolean)
        MyBase.AutoScroll = False
    End Set
End Property
```

Néanmoins, si cela masque le membre en apparence, il existe toujours. Pour être sur qu'elle ne soit pas utilisée, vous devez donc la marquer aussi avec l'attribut Obsolete en forçant le déclenchement d'une erreur. La déclaration précédente devrait être :

```
<Browsable(False), EditorBrowsable(EditorBrowsableState.Never),
Obsolete("propriété indisponible", True)> Overrides Property AutoScroll()
As Boolean
```

Comme vous le voyez, il va y avoir une utilisation importante des attributs. En effet, c'est par eux qu'on définit une partie du fonctionnement des composants.

Commençons à construire la logique du contrôle. Je vais ajouter trois variables internes pour gérer mon contrôle tel que :

```
#Region "variables"
    Private m_IsOpen As Boolean = True
    Private m_OpenHeight As Integer = 176
    Private m_CloseHeight As Integer = 20
#End Region
```

Notez que je leur attribue une valeur étant mes propriétés par défaut.

Ajout de membres

Comme nous l'avons vu, ce contrôle utilisateur est une classe. Je procède donc de façon tout à fait habituelle. J'ajoute quelques propriétés soit pour l'apparence du bouton, soit pour le fonctionnement du contrôle :

```
#Region "Membres ajoutés"
    <Browsable(True), Description("Etat ouvert ou fermé de la zone"),
Category("behavior")> _
    Public Property IsOpen() As Boolean
        Get
            Return m_IsOpen
        End Get
```

```
        Set(ByVal Value As Boolean)
            m_IsOpen = Value
            If Value Then
                Me.Height = m_OpenHeight
            Else
                Me.Height = m_CloseHeight
            End If
        End Set
    End Property

    <Browsable(True), Description("Taille Mini de la zone"),
Category("Appearance")> _
    Public Property CloseHeight() As Integer
        Get
            Return m_CloseHeight
        End Get
        Set(ByVal Value As Integer)
            If Value < Me.ButtonHeight + 4 Then Value = Me.ButtonHeight +
4
            m_CloseHeight = Value
        End Set
    End Property

    <Browsable(True), Description("Taille de la zone agrandie"),
Category("Appearance")> _
        Public Property OpenHeight() As Integer
        Get
            Return m_OpenHeight
        End Get
        Set(ByVal Value As Integer)
            If Me.CloseHeight > Value Then Value = Me.CloseHeight
            m_OpenHeight = Value
        End Set
    End Property

    <Browsable(True), Description("Couleur de fond du bouton"),
Category("Appearance")> _
    Public Property ButtonBackColor() As Color
        Get
            Return Me.Button1.BackColor
        End Get
        Set(ByVal Value As Color)
            Me.Button1.BackColor = Value
        End Set
    End Property

    <Browsable(True), Description("Couleur de texte du bouton"),
Category("Appearance")> _
    Public Property ButtonForeColor() As Color
        Get
            Return Me.Button1.ForeColor
        End Get
        Set(ByVal Value As Color)
            Me.Button1.ForeColor = Value
        End Set
    End Property
```

```
    <Browsable(True), Description("Hauteur du bouton"),
Category("Appearance")> _
    Public Property ButtonHeight() As Integer
        Get
            Return Me.Button1.Height
        End Get
        Set(ByVal Value As Integer)
            Me.Button1.Height = Value
            If Me.CloseHeight < Me.Button1.Height + 4 Then Me.CloseHeight
= Me.Button1.Height + 4
        End Set
    End Property

    Public Property Text() As String
        Get
            Return Me.Button1.Text
        End Get
        Set(ByVal Value As String)
            Me.Button1.Text = Value
        End Set
    End Property

#End Region
```

Là encore, vous voyez que j'utilise pour chaque propriété trois attributs :

- Browsable : Affiche la propriété dans la fenêtre propriétés du contrôle
- Description : Affiche le texte en paramètre dans la fenêtre description
- Catégory : Défini la catégorie de la propriété

Modification de membres existants

Je vais maintenant modifier quelques membres existants pour obtenir un comportement spécifique.

En résumé, je vais modifier la méthode ToString pour pouvoir renvoyer le texte du bouton. Je vais aussi modifier le comportement de la méthode OnControlAdded afin de redimensionner mon contrôle si les contrôles ajoutés sortent de ses limites (puisque le contrôle ne gère pas l'Autoscroll). Enfin je vais modifier la méthode OnPaint pour tracer la bordure 3D qui simulera le contrôle GroupBox.

Enfin je vais gérer le re-dimensionnement afin de modifier les propriétés OpenHeight et CloseHeight dynamiquement si la hauteur change.

```
#Region "Membres modifiés"

    Public Overrides Function ToString() As String
        Return Me.Button1.Text
    End Function

    Protected Overrides Sub OnControlAdded(ByVal e As
System.Windows.Forms.ControlEventArgs)
        If Not e.Control Is Me.Button1 Then
            Me.Controls.Add(e.Control)
            If e.Control.Location.Y + e.Control.Height > Me.Height + 2
Then Me.Height = e.Control.Location.Y + e.Control.Height + 2
            If e.Control.Location.X + e.Control.Width > Me.Width + 2 Then
Me.Width = e.Control.Location.X + e.Control.Width + 2
            Me.Refresh()
        End If
    End Sub
```

```
    Protected Overrides Sub OnPaint(ByVal e As
System.Windows.Forms.PaintEventArgs)
        e.Graphics.Clear(System.Drawing.SystemColors.Control)
        MyBase.OnPaint(e)
        System.Windows.Forms.ControlPaint.DrawBorder3D(e.Graphics, 0,
CInt(Me.ButtonHeight / 2), Me.Width, CInt(Me.Height - Me.ButtonHeight /
2))
    End Sub

    Protected Overrides Sub OnSizeChanged(ByVal e As System.EventArgs)
        MyBase.OnSizeChanged(e)
        If Me.DesignMode Then
            If Me.IsOpen Then
                If Me.Size.Height > Me.CloseHeight Then
                    Me.OpenHeight = Me.Size.Height
                Else
                    Me.OpenHeight = Me.CloseHeight
                    Me.Height = Me.OpenHeight
                End If
            Else
                If Me.Size.Height > Me.ButtonHeight + 4 Then
                    Me.CloseHeight = Me.Size.Height
                Else
                    Me.CloseHeight = Me.ButtonHeight + 4
                    Me.Height = Me.CloseHeight
                End If
            End If
        End If
        Me.Refresh()

    End Sub

#End Region
```

Notez que j'utilise le test *Me.DesignMode*. Celui-ci indique je souhaite que le redimensionnement n'influe que dans le cas où le contrôle est en mode conception. Comme nous le verrons plus loin, on peut maintenant gérer le contrôle dans son mode conception et dans son mode exécution.

Ajout d'évènements

Nous avons déjà vu dans les parties précédentes la création d'évènements. Pour mémoire, je rappelle ici le schéma.

- Création, si besoin est, d'une classe dérivant de System.EventArgs.
- Définition de l'évènement
- Déclaration d'un délégué ayant la signature de l'évènement
- Levée de l'évènement

Ce schéma est assez simple. Par convention, lorsqu'on code un composant autonome héritable, on place la levée de l'évènement dans une procédure portant comme nom « OnNomEvenement », avec une portée Protected. Pourquoi cela ?

Pour le développeur qui souhaite reprendre le composant par héritage, une telle approche lui permet de savoir exactement dans qu'elle méthode a lieu la levée d'un évènement spécifique. De plus, cette centralisation fait que même si plusieurs causes peuvent déclencher un évènement, la seule substitution de cette méthode permet une modification homogène du comportement. Enfin, le développeur garde la possibilité de déclencher ou non la levée de l'évènement d'origine en faisant appel à la méthode de la classe de base.

Digressons un peu. Bien que je ne sois pas un fou des conventions « universelles » de codage, reconnaissons que certaines d'entres elles sont souvent indispensables à une lisibilité correcte du code. Ainsi suffixer vos arguments par 'Args', vos attributs par 'Attribute' et ainsi de suite est un minimum. De même, dans une conception objet, et donc, de manipulation d'héritage, respecter un certain nombre de convention simplifie le travail de récupération.

Dans notre cas, nous voulons donc gérer l'évènement ButtonClick de notre contrôle. Comme dans le cas des évènements click, il n'y a pas besoin d'arguments spécifiques, nous allons simplement utiliser System.EventArgs comme argument. Dans ma classe je vais donc déclarer mon évènement tel que :

```
Public Event ButtonClick(ByVal sender As Object, ByVal e As
System.EventArgs)
```

En dehors de ma classe, je vais déclarer le délégué correspondant :

```
Delegate Sub ButtonClickEventHandler(ByVal sender As Object, ByVal e As
System.EventArgs)
```

Enfin je vais ajouter le membre OnBouttonClick qui lèvera l'évènement :

```
Protected Overridable Sub OnButtonClick(ByVal e As System.EventArgs)
    Me.IsOpen = Not Me.IsOpen
    RaiseEvent ButtonClick(Me, New EventArgs)
End Sub
```

Sans oublier d'appeler ce membre lors d'un click sur le bouton de mon contrôle :

```
Private Sub Button1_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Button1.Click
    Me.OnButtonClick(e)
End Sub
```

Extension du mode design

Nous avons fini notre composant. Il est fonctionnel en l'état et pourtant ce n'est pas satisfaisant. En effet, si nous posons sur lui deux zones de texte en mode conception, celles-ci prennent comme parent notre formulaire au lieu du contrôle. Il est bien sur possible d'aller modifier le code généré pour corriger cet état de fait, mais ce n'est pas aisé.

Cette partie peut nous mener assez loin, aussi vais-je seulement survoler la question. Eventuellement j'approfondirais la question dans un article pour la version 2005.

Donc quel est le problème ?

La plupart des contrôles ont un fonctionnement différent selon qu'on est en mode conception (Design) ou exécution (runtime). Par exemple, on peut sélectionner un contrôle déposé sur un formulaire en mode Design ce que l'on ne peut pas faire à l'exécution sans développer le code correspondant. Sur le même principe, notre composant n'a pas de fonctionnalités en mode Design tant qu'elles n'ont pas été développées. Ce développement est assez spécifique et, uniquement utile, si vous développez vos propres composants, par conséquent, je ne vais vous montrer ici que l'approche.

Pour gérer le mode Design d'un composant, on utilise un Designer appelé aussi *concepteur personnalisé*.

Il en existe plusieurs types de base selon les cas. Dans le nôtre, il s'agit d'implémenter la capacité d'être conteneur de contrôles en phase de conception. La classe de base est ParentControlDesigner.

Dès lors, je vais créer une classe MaxBoxDesigner qui hérite de ParentControlDesigner. Cette classe commencerait telle que :

```
Public Class MaxBoxDesigner
    Inherits Windows.Forms.Design.ParentControlDesigner

    Private WithEvents ChangeServ As IComponentChangeService
    Private WithEvents SelectServ As ISelectionService
    Private MonControl As MaxBox

    Public Overrides Sub Initialize(ByVal component As
System.ComponentModel.IComponent)
        MyBase.Initialize(component)
```

```
        ChangeServ =
CType(Me.GetService(GetType(IComponentChangeService)),
IComponentChangeService)
        SelectServ = CType(Me.GetService(GetType(ISelectionService)),
ISelectionService)
        MonControl = CType(component, MaxBox)
    End Sub

    Private Sub ChangeServ_ComponentAdding(ByVal sender As Object, ByVal
e As System.ComponentModel.Design.ComponentEventArgs) Handles
ChangeServ.ComponentAdding
        If MonControl.IsOpen = False Then e.Component.Dispose()
    End Sub

    Protected Overrides Function GetHitTest(ByVal point As
System.Drawing.Point) As Boolean
        Dim ControlContenu As Control
        Dim Rect As Rectangle

        point = MonControl.PointToClient(point)
        For Each ControlContenu In MonControl.Controls
            Rect = ControlContenu.Bounds
            If Rect.Contains(point) Then
                Return True
                Exit Function
            End If
        Next
        Return False
    End Function

    Private Sub SelectServ_SelectionChanged(ByVal sender As Object, ByVal
e As System.EventArgs) Handles SelectServ.SelectionChanged
        MonControl.OnSelectionChanged()
        Control.Invalidate()
    End Sub

    Public Overrides ReadOnly Property SelectionRules() As
System.Windows.Forms.Design.SelectionRules
        Get
            Return SelectionRules.LeftSizeable Or
SelectionRules.RightSizeable Or _
                        SelectionRules.Moveable Or SelectionRules.Visible
        End Get
    End Property

    Private Sub OnComponentChanged(ByVal sender As Object, ByVal ce As
ComponentChangedEventArgs)
        Control.Invalidate()
    End Sub

    Protected Overrides Sub OnPaintAdornments(ByVal pe As
System.Windows.Forms.PaintEventArgs)

        If Not SelectServ Is Nothing Then
            Dim SelControl As Control =
CType(SelectServ.PrimarySelection, Control)
```

```
            If Not SelControl.GetType Is MonControl.GetType Then
                Dim g As Graphics = pe.Graphics
                Dim handle As Rectangle = New
Rectangle(SelControl.Location, SelControl.Size)
                handle.Offset(-1, -1)
                handle.Inflate(2, 2)
                ControlPaint.DrawGrabHandle(g, handle, True, True)
            End If
        End If

    End Sub

End Class
```

Pour faire au plus simple, disons que je vais mettre en œuvre des services pour pouvoir manipuler mon contrôle au moment du Design. Plus je vais vouloir donner de fonctionnalité à mon contrôle, plus le code du Designer va être complexe. En l'état, mon contrôle va bien être le parent des contrôles que je vais déposer sur lui, il va bien me permettre de les sélectionner, mais je ne pourrais pas les déplacer à l'aide de la souris. Pour réaliser cela, il faudra que je gère ses fonctions de glisser / déplacer.

Notez que, pour associer le contrôle à son Designer, j'utilise un attribut DesignerAttribute ce qui me donne une ligne de déclaration :

```
<Designer(GetType(MaxBoxDesigner))> Public Class MaxBox
```

Notez aussi que je dois aussi implémenter de nouvelles fonctions dans le contrôle pour que le Designer agisse correctement. Dans le cas du Designer donné ci-dessus, je devrais ajouter au contrôle MaxBox le code :

```
#Region "Modif designer"
    Protected Overrides Sub OnMouseDown(ByVal e As
System.Windows.Forms.MouseEventArgs)
        Dim Ctrl As Control
        Dim Rect As Rectangle
        Dim SelServ As ISelectionService
        Dim Tableau As ArrayList

        If DesignMode Then
            For Each Ctrl In Me.Controls
                Rect = Ctrl.Bounds
                If Rect.Contains(e.X, e.Y) Then
                    SelServ =
CType(Me.GetService(GetType(ISelectionService)), ISelectionService)
                    Tableau = New ArrayList
                    Tableau.Add(Ctrl)
                    SelServ.SetSelectedComponents(Tableau)
                    Exit For
                End If
            Next
        End If
        MyBase.OnMouseDown(e)
    End Sub

    Friend Sub OnSelectionChanged()
        Dim Ctrl As Control
        Dim NewSelectControl As Control = Nothing
        Dim SelServ As ISelectionService

        SelServ = CType(Me.GetService(GetType(ISelectionService)),
ISelectionService)
```

```
        For Each Ctrl In Me.Controls
            If SelServ.PrimarySelection Is Ctrl Then
                NewSelectControl = Ctrl
                Exit For
            End If
        Next
        If Not NewSelectControl Is SelectedControl Then
            SelectedControl = NewSelectControl
            Invalidate()
        End If
    End Sub

#End Region
```

Lorsqu'au cours de mon apprentissage de VB.NET j'ai commencé à me frotter aux contrôles utilisateurs, j'ai été surpris par l'extraordinaire complexité de la création de ceux-ci. Tout au moins lorsqu'on part d'un concepteur assez simple.

Glisser - Déplacer

A la différence de VB 6, il n'y a plus deux méthodes à utiliser selon qu'on soit ou non au sein du même formulaire pour l'opération. Autrement dit, c'est une même technique qui gère les opérations de glisser – déplacer quelque soit la source et la cible. Cependant, il y a des cas où le code sera un peu plus élaboré selon ce que l'on transmet. Nous allons voir quelques exemples qui fourniront une bonne révision de ce que nous avons déjà vu et qui nous permettront de manipuler le presse-papiers.

Le presse-papiers en VB.NET

Vous n'allez pas être tellement dépaysé car le fonctionnement est identique si ce n'est qu'il n'existe plus qu'une méthode pour mettre des données dans le presse-papiers (SetDataObject) et une pour les récupérer (GetDataObject). La méthode GetFormat de VB6 qui permettait d'identifier le format des données du presse-papiers s'appelle maintenant GetDataPresents. Comme le presse-papiers peut contenir plus d'un élément ou, un seul élément stocké sur plusieurs formes, on peut récupérer l'objet DataObject du presse-papiers pour le tester.

Donc, en VB6, on avait des codes du style :

```
Private Sub Command1_Click()
Clipboard.SetText Me.Text1.Text
If Clipboard.GetFormat(vbCFText) Then
    Me.Text2.Text = Clipboard.GetText(vbCFText)
End If
End Sub

Private Sub Command2_Click()
Clipboard.SetData Me.Picture1.Picture
If Clipboard.GetFormat(vbCFBitmap) Then
    Me.Picture2.Picture = Clipboard.GetData(vbCFBitmap)
End If
End Sub
```

qui deviennent dans VB.NET.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click

    Clipboard.SetDataObject(Me.TextBox1.Text)
    Dim objClip As IDataObject = Clipboard.GetDataObject
    If objClip.GetDataPresent(DataFormats.Text) Then
        TextBox2.Text = objClip.GetData(DataFormats.Text).ToString()
    End If

End Sub

Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button2.Click

    Clipboard.SetDataObject(Me.PictureBox1.Image)
    Dim objClip As IDataObject = Clipboard.GetDataObject
    If objClip.GetDataPresent(DataFormats.Bitmap) Then
        Me.PictureBox2.Image = CType(objClip.GetData(DataFormats.Bitmap),
Bitmap)
    End If

End Sub
```

N'oubliez pas toutefois que le presse-papiers est un élément commun à toutes les applications, dès lors les données peuvent être vulnérables et il convient généralement d'appliquer des tests si celles-ci présentent un caractère sensible.

A ce propos, la classe UIPermission gère des permissions pour l'accès au presse-papiers. Si votre code doit s'exécuter dans une une zone de moindre privilège, faites attention car l'extraction de données du presse-papiers peut vous être refusé.

Fonctionnement général

Les opérations de glisser – déplacer suivent sensiblement toujours la même procédure :

- Un évènement d'initialisation : c'est généralement l'évènement MouseDown du contrôle source mais certains contrôles implémentent un évènement spécifique (ItemDrag par exemple)
- Des évènements de déplacement qui n'interviennent pas véritablement dans l'opération sauf pour la faire échouer mais qui gèrent l'effet visuel.
- Un évènement de fin d'opération, DragDrop qui gère le déplacement proprement dit.

La méthode d'appel du démarrage de l'opération n'est plus Drag comme en VB6 mais DoDragDrop.

Elle suit la syntaxe :

```
Public Function DoDragDrop(ByVal data As Object, ByVal allowedEffects As
DragDropEffects) As DragDropEffects
```

Il faut donc forcément passer un objet à la méthode même si, in fine, iln'est pas toujours nécessaire comme nous le verrons dans le troisième exemple. La liste des effets est définie par :

Nom de membre	Description	Valeur
Scroll	Le défilement est sur le point de commencer ou est en cours dans la cible de déplacement.	-2147483648
All	Les données sont copiées, supprimées de la source de glissement et parcourues dans la cible de déplacement.	-2147483645
None	La cible de déplacement n'accepte pas les données.	0
Copy	Les données sont copiées dans la cible de déplacement.	1
Move	Les données issues de la source de glissement sont déplacées vers la cible de déplacement.	2
Link	Les données issues de la source de glissement sont liées à la cible de déplacement.	4

Exemples d'opération glisser – déplacer

Voyons maintenant trois exemples classiques qui nous permettront de couvrir plusieurs scénarios.

Copie de données

Dans cet exemple, nous allons copier une valeur d'un élément de zone de liste dans une zone de texte. Il s'agit donc d'une opération de copie puisque l'élément source n'est pas modifié. Ce n'est pas l'élément qui est en jeu mais uniquement le texte de celui-ci.

Reprenons les étapes du cas général :

Initialisation

```
Private Sub listBox1_MouseDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles ListBox1.MouseDown

    Dim lb As ListBox = CType(sender, ListBox)
    Dim pt As New Point(e.X, e.Y)
    Dim index As Integer = lb.IndexFromPoint(pt)

    If index >= 0 Then lb.DoDragDrop(lb.Items(index).ToString(),
DragDropEffects.Copy)

End Sub
```

Ce code va donc initialiser l'opération. Le calcul de l'index sert à récupérer la position de l'élément étant sous le curseur lors de l'évènement MouseDown. L'objet passé est le texte de l'élément pour un effet visuel de copy.

Mouvement

```
Private Sub TextBox1_DragEnter(ByVal sender As Object, ByVal e As
System.Windows.Forms.DragEventArgs) Handles TextBox1.DragEnter

    If (e.Data.GetDataPresent(DataFormats.Text)) Then
        e.Effect = DragDropEffects.Copy
    Else
        e.Effect = DragDropEffects.None
    End If

End Sub
```

En fait, dans ce cas je ne gère l'effet visuel que lors de l'entrée dans le contrôle cible. Au survol de la feuille, j'aurai un curseur d'interdiction de déplacement puisque je ne gère pas les évènements Drag du formulaire. Donc, à l'entrée dans ma zone de texte, je vérifie que la donnée présente est bien du texte, si c'est le cas, le curseur de copie s'affiche, sinon c'est le curseur d'interdiction.

Revenons sur le cas du formulaire. Je pourrais gérer un effet visuel lors du survol du formulaire. Ceci serait un peu risqué car l'utilisateur pourrait croire que le formulaire est une cible valide ce qui n'est pas le cas. Mais supposons que je veuille le faire quand même. J'écris alors :

```
Private Sub Form1_DragOver(ByVal sender As Object, ByVal e As
System.Windows.Forms.DragEventArgs) Handles MyBase.DragOver
    e.Effect = DragDropEffects.Move
End Sub
```

Bien que correct, ce code ne fonctionnera pas. De fait, les effets autorisés lors d'une opération de glisser déplacer sont fixées lors de l'appel à DoDragDrop. Or mon appel est :

```
lb.DoDragDrop(lb.Items(index).ToString(), DragDropEffects.Copy)
```

Cela revient à dire que seul l'effet Copy est autorisé. Soit je dois mettre DragDropEffects.Copy dans mon évènement DragOver, soit je dois modifier mon appel DoDragDrop tel que :

```
If index >= 0 Then lb.DoDragDrop(lb.Items(index).ToString(),
DragDropEffects.Copy Or DragDropEffects.Move)
```

Notez que l'argument DragEventArgs contient une propriété AllowedEffect qui permet de tester les effets visuels autorisés par le code.

Fin d'opération

Le code est très simple

```
Private Sub TextBox1_DragDrop(ByVal sender As Object, ByVal e As
System.Windows.Forms.DragEventArgs) Handles TextBox1.DragDrop

    If e.Data.GetDataPresent(DataFormats.Text) Then TextBox1.Text =
e.Data.GetData(DataFormats.Text).ToString

End Sub
```

Déplacement d'élément

Ce deuxième exemple est plus spécifique. Il s'agit de déplacer un élément de contrôle dans ce même contrôle. Dans notre cas, nous allons gérer le déplacement d'un élément de TreeView dans le même TreeView.

Le contrôle TreeView expose un évènement ItemDrag pour gérer le début de l'opération. Dans cet exemple nous allons gérer les évènements qui vont se produire successivement dans cet ordre :

- ItemDrag
- DragEnter
- DragOver
- DragDrop

Le début du code va être similaire à l'exemple précédent :

```
Public Sub TreeView1_ItemDrag(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.ItemDragEventArgs) Handles TreeView1.ItemDrag
    DoDragDrop(e.Item, DragDropEffects.Move)

End Sub

Public Sub TreeView1_DragEnter(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.DragEventArgs) Handles TreeView1.DragEnter

    If e.Data.GetDataPresent("System.Windows.Forms.TreeNode", True) Then
        e.Effect = DragDropEffects.Move
    Else
        e.Effect = DragDropEffects.None
    End If

End Sub
```

Par contre le code du DragOver va être un peu plus complexe :

```
Public Sub TreeView1_DragOver(ByVal sender As System.Object, ByVal e As
DragEventArgs) Handles TreeView1.DragOver

    If e.Data.GetDataPresent("System.Windows.Forms.TreeNode", True) =
False Then Exit Sub
    Dim pt As Point = CType(sender, TreeView).PointToClient(New
Point(e.X, e.Y))
    Dim NodeCible As TreeNode = Me.TreeView1.GetNodeAt(pt)
    Dim NodeSource As TreeNode =
CType(e.Data.GetData("System.Windows.Forms.TreeNode"), TreeNode)
```

```
    If Not (Me.TreeView1.SelectedNode Is NodeCible) Then
Me.TreeView1.SelectedNode = NodeCible
    Do Until NodeCible Is Nothing
        If NodeCible Is NodeSource Then
            e.Effect = DragDropEffects.None
            Exit Sub
        End If
        NodeCible = NodeCible.Parent
    Loop
    e.Effect = DragDropEffects.Move

End Sub
```

Pourquoi ne gère-t-on pas simplement l'effet juste après un test du DataObject ? Pour que le Drag & Drop fonctionne.

Puisque nous sommes dans le même contrôle, il faut que le mouvement de la souris serve aussi à détecter en temps réel le nœud cible. Pour cela, nous allons donc utiliser l'évènement DragOver et la méthode GetNodeAt de l'objet TreeView pour sélectionner le nœud sous le curseur. Dans le même temps, nous allons faire une boucle de test pour vérifier que le nœud en cours de déplacement n'a pas pour cible lui-même ou un de ses enfants. Si tel était le cas, le déplacement ne serait pas possible sans casser l'arbre.

Le code de fin d'opération sera :

```
Public Sub TreeView1_DragDrop(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.DragEventArgs) Handles TreeView1.DragDrop

    If e.Data.GetDataPresent("System.Windows.Forms.TreeNode", True) =
False Then Exit Sub
    Dim NodeSource As TreeNode =
CType(e.Data.GetData("System.Windows.Forms.TreeNode"), TreeNode)
    Dim NodeCible As TreeNode = Me.TreeView1.SelectedNode
    Dim VerifParent As TreeNode = NodeCible
    Do Until VerifParent Is Nothing
        If VerifParent Is NodeSource Then Exit Sub
        VerifParent = VerifParent.Parent
    Loop
    NodeSource.Remove()
    If NodeCible Is Nothing Then
        Me.TreeView1.Nodes.Add(NodeSource)
    Else
        NodeCible.Nodes.Add(NodeSource)
    End If
    NodeSource.EnsureVisible()
    Me.TreeView1.SelectedNode = NodeSource

End Sub
```

Déplacement de contrôles

Nous allons voir un cas un peu plus complexe où le but est de faire un TextBox déplaçable à l'exécution. Abordons la problématique ensemble.

Le code de démarrage de l'action pourrait toujours être de la même forme, c'est-à-dire :

```
Private Sub TextBox3_MouseDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles TextBox3.MouseDown

    Me.TextBox3.DoDragDrop(Me.TextBox3, DragDropEffects.Move)
End Sub
```

On créerait alors un évènement DragEnter et DragDrop tel que :

```
Private Sub Form1_DragEnter(ByVal sender As Object, ByVal e As
System.Windows.Forms.DragEventArgs) Handles MyBase.DragEnter

    e.Effect = DragDropEffects.Move

End Sub

Private Sub Form1_DragDrop(ByVal sender As Object, ByVal e As
System.Windows.Forms.DragEventArgs) Handles MyBase.DragDrop

    With Me.TextBox3
        .Location = Me.PointToClient(New Point(e.X, e.Y))
        .Visible = True
    End With

End Sub
```

Ce code est très simple, mais vous voyez très certainement le problème. Dans ce cas, je ne teste pas la nature du contenu puisque je ne l'ai pas dans ma liste de DataFormats. Pire, je ne le teste pas non plus lors du déplacement puisque je ne sais pas extraire un objet dont je ne connais pas le format. Evidemment, s'il s'agit de la seule opération de déplacement autorisée dans la feuille, c'est suffisant. Mais si tel n'est pas le cas, mon code fera n'importe quoi.

Heureusement tout cela est faux. Je peux utiliser n'importe quel type tant pour GetDataPresent que pour GetData et le code correct sera alors :

```
Private Sub TextBox3_MouseDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles TextBox3.MouseDown
    Me.TextBox3.DoDragDrop(Me.TextBox3, DragDropEffects.Move)
End Sub

Private Sub Form1_DragEnter(ByVal sender As Object, ByVal e As
System.Windows.Forms.DragEventArgs) Handles MyBase.DragEnter

    If e.Data.GetDataPresent("System.Windows.Forms.TextBox") Then
        e.Effect = DragDropEffects.Move
    End If

End Sub

Private Sub Form1_DragOver(ByVal sender As Object, ByVal e As
System.Windows.Forms.DragEventArgs) Handles MyBase.DragOver
    If e.Data.GetDataPresent("System.Windows.Forms.TextBox") Then
e.Effect = DragDropEffects.Move
End Sub

Private Sub Form1_DragDrop(ByVal sender As Object, ByVal e As
System.Windows.Forms.DragEventArgs) Handles MyBase.DragDrop

    If e.Data.GetDataPresent("System.Windows.Forms.TextBox") Then
        Dim MaTextBox As Windows.Forms.TextBox =
CType(e.Data.GetData("System.Windows.Forms.TextBox"), TextBox)
        With MaTextBox
            .Location = Me.PointToClient(New Point(e.X, e.Y))
            .Visible = True
        End With
    End If
End Sub
```

L'impression

S'il y a bien une constante dans les versions de Visual Basic, c'est le travail de gestion de l'impression. Admettons qu'il y a quelques mieux dans VB.NET mais ce n'est pas pour cela que cela va être simple. Dans VB.NET vous ne gérez pas un objet Printer mais un contrôle PrintDocument. La bonne nouvelle, par contre, est qu'il existe maintenant un contrôle gérant *l'aperçu avant impression* qui évite de devoir passer une rame complète pour tester son code. Je ne vais pas vous faire un sujet complet ici sur l'impression, mais nous allons voir les fondamentaux.

L'espace de nom qui gère l'impression est **System.Drawing.Printing**.

L'imprimante

Dans la majorité des cas, un gestionnaire d'impression est indépendant de l'imprimante choisie. Pour cela, il est nécessaire de récupérer les informations de l'imprimante. Par ailleurs, il est de plus souvent utile de proposer le choix de l'imprimante à l'utilisateur. VB.NET vous propose deux contrôles standards pour le choix et/ou le réglage de l'imprimante : PrintDialog et PageSetupDialog. Pour ma part, je préfère le contrôle PageSetupDialog qui permet d'accéder aux propriétés de mise en page mais aussi à la définition de l'imprimante. La manipulation de ces contrôles est assez aisée, mais vous ne devez pas oublier qu'ils demandent une instance d'un objet PrintDocument. Ceci implique que les sélections réalisées dans les boites de dialogues ne ciblent que le document en cours et n'affectent pas le paramétrage de l'imprimante pour d'autres documents et/ou applications.

Un exemple simple de récupération d'informations serait par exemple:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    Dim Doc As New PrintDocument
    With Me.PageSetupDialog1
        .Document = Doc
        .ShowDialog(Me)
    End With
    With Me.TextBox1
        .Text = Doc.PrinterSettings.PrinterName & vbCrLf
        .Text = .Text & Doc.PrinterSettings.Copies.ToString & vbCrLf
        .Text = .Text & Doc.PrinterSettings.Collate.ToString & vbCrLf
        .Text = .Text & Doc.DefaultPageSettings.Color.ToString & vbCrLf
        .Text = .Text & Doc.DefaultPageSettings.Landscape.ToString &
vbCrLf
        .Text = .Text & Doc.DefaultPageSettings.PaperSource.ToString &
vbCrLf
    End With
End Sub
```

L'impression

Impression d'un formulaire

La méthode PrintForm n'existe plus en VB.NET. Elle est remplaçable par une impression de l'image du formulaire. Le code est d'ailleurs donné dans l'aide au chapitre « Code : impression d'un Windows Form ». Une technique couramment utilisée en VB6 consistait à manipuler le formulaire le temps de l'impression pour ne pas avoir à gérer l'impression. Cette approche est toujours possible mais, comme en VB 6, je ne la recommande pas.

Le problème géométrique

J'ai écris de nombreux gestionnaire d'impression en VB 6. Bien que ce soit assez fastidieux, ce n'est jamais complexe dès qu'on a réglé les deux problèmes principaux : l'unité de mesure et la zone d'impression.

L'unité de mesure

C'est toujours là qu'on se plante lorsqu'on débute l'écriture d'un gestionnaire d'impression. Assez rapidement, on mélange millimètre, pouce et pixel avec un bonheur inégal. On se lance alors dans des conversions audacieuses ce qui finit par engendrer un gestionnaire d'impression long comme l'œuvre de Zola pour imprimer trois cadres et deux textes. Il y a donc une règle qui doit être votre unique base de travail : la propagation du choix de l'unité au gestionnaire tout entier.

Bien que l'immonde Twips ait disparu dans VB.NET, il reste encore bien des occasions de se tromper. Par défaut, les contrôles et les paramètres d'impression gèrent des centièmes de pouces, alors que les méthodes graphiques invoquées renvoient des pixels. Il est donc impératif d'harmoniser cela. Comme nous allons le voir dans l'exemple, VB.NET donne accès à des convertisseurs assez explicites qui rendent le code nettement plus lisible. N'oubliez cependant jamais que, chaque objet devant être imprimé, doit correspondre au bon système d'unité.

La zone d'impression

C'est le deuxième loup de cette jungle car elle demande une compréhension correcte de ce qu'est la zone d'impression. Cette expression recouvre plusieurs concepts dans lesquels on se mélange généralement les pinceaux. Par ordre de taille nous avons successivement :

La taille de la page
La zone imprimable
La zone d'impression

La taille de la page, tout le monde se représente bien ce que c'est. Pour le format A4 par exemple, c'est 297x210 mm.

La zone imprimable est autrement plus complexe. Elle est inhérente à chaque imprimante et n'est pas forcément centrée, ou si vous préférez, les marges ne sont pas forcément symétriques. L'obtention de cette zone n'est pas évidente.

La zone d'impression, c'est la zone délimitée par les marges par défaut du PageSetup ou par les marges que vous avez fixées le cas échéant.

Pour bien comprendre le principe, imaginons le code suivant :

```
    Private Declare Function GetDeviceCaps Lib "gdi32" (ByVal hdc As
IntPtr, ByVal nIndex As Integer) As Integer

    Private Const HORZRES As Short = 8
    Private Const VERTRES As Short = 10
    Private Const PHYSICALWIDTH As Short = 110
    Private Const PHYSICALHEIGHT As Short = 111
    Private Const PHYSICALOFFSETX As Short = 112
    Private Const PHYSICALOFFSETY As Short = 113

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    Me.PageSetupDialog1.ShowDialog()
    Me.PrintPreviewDialog1.ShowDialog()
End Sub

Private Sub PrintDocument1_PrintPage(ByVal sender As Object, ByVal e As
System.Drawing.Printing.PrintPageEventArgs) Handles
PrintDocument1.PrintPage
```

```
    Const ConvTomm As Single = 25.4
    Dim HDC As IntPtr = e.Graphics.GetHdc
    Dim MargeGauche, MargeDroite, MargeHaute, MargeBasse As Single
    Dim Hauteur, Largeur, HImprim, LImprim As Single
    MargeGauche = GetDeviceCaps(HDC, PHYSICALOFFSETX)
    MargeHaute = GetDeviceCaps(HDC, PHYSICALOFFSETY)
    LImprim = GetDeviceCaps(HDC, HORZRES)
    HImprim = GetDeviceCaps(HDC, VERTRES)
    Hauteur = GetDeviceCaps(HDC, PHYSICALHEIGHT)
    Largeur = GetDeviceCaps(HDC, PHYSICALWIDTH)
    e.Graphics.ReleaseHdc(HDC)
    MargeGauche = MargeGauche * ConvTomm / e.Graphics.DpiX
    MargeHaute = MargeHaute * ConvTomm / e.Graphics.DpiY
    LImprim = LImprim * ConvTomm / e.Graphics.DpiX
    HImprim = HImprim * ConvTomm / e.Graphics.DpiY
    Largeur = Largeur * ConvTomm / (e.Graphics.DpiX)
    Hauteur = Hauteur * ConvTomm / (e.Graphics.DpiY)
    MargeDroite = Largeur - LImprim - MargeGauche
    MargeBasse = Hauteur - HImprim - MargeHaute
    e.Graphics.DrawRectangle(Pens.Blue, e.PageBounds)
    e.Graphics.DrawRectangle(Pens.Green, e.MarginBounds)
    e.Graphics.PageUnit = GraphicsUnit.Millimeter
    e.Graphics.DrawRectangle(New Pen(Color.Red, 1),
Rectangle.Ceiling(RectangleF.FromLTRB(MargeGauche + 1, MargeHaute + 1,
LImprim - 2, HImprim - 2)))
    e.Graphics.Dispose()

End Sub
```

Je n'ai pas trouvé dans le Framework une classe permettant de lire les contraintes de l'imprimante. Pour atteindre ces paramètres physiques de l'imprimante, j'ai donc appelé une API Windows, GetDeviceCaps, tout comme je le faisais en VB6.

J'utilise le contrôle PrintPreviewDialog pour visualiser mon résultat. Je vois trois rectangles imbriqués, l'un représentant les limites de la page, l'autre les limites de la zone imprimable et enfin les limites de la zone d'impression. Notez que si l'affichage est correct, l'impression sur papier ne donnerait pas le même résultat. Cela est du au fait que le rectangle PageBounds, bien tracé sur le bord de la feuille dans l'aperçu avant impression, sera tracé aux limites de la zone imprimable. Donc, le point (0,0) de la feuille est déterminé par les marges physiques de l'imprimante.

Là, je pense que je vous ai suffisamment embrouillé pour qu'un exemple soit devenu indispensable. Nous le verrons un peu plus loin.

Fonctionnement général

A la base vous avez obligatoirement une instance de PrintDocument. Tous vos contrôles de configuration ou d'impression attendront ce PrintDocument pour pouvoir fonctionner puisque l'impression dans VB.NET passe obligatoirement par lui.

Normalement, si votre programme laisse le choix à l'utilisateur, vous devez lui afficher une boite de dialogue de choix d'imprimante et/ou de paramétrage. Cependant, vous pouvez configurer celle-ci afin d'interdire certaines options. Tout peut évidemment être géré par le code.

Ne perdez pas de vue que les boites de dialogues ont un bouton "Annuler" et qu'il convient donc de gérer complètement la réponse de l'utilisateur.

L'impression va alors se déclencher que ce soit pour un aperçu ou pour une impression. Il y a pour cela appel à un objet PrintController. Généralement on ne l'utilise pas explicitement, mais il peut être utile de le dériver pour des impressions un peu complexes.

Ce PrintController va gérer une séquence d'appel vers les évènements de PrintDocument. C'est généralement ceux-ci que l'on intercepte.

Dans la plupart des cas, il suffit d'écrire le code dans l'évènement PrintPage. Cependant, il peut parfois être intéressant d'utiliser BeginPrint pour vérifier la configuration demandée par l'utilisateur ou QueryPageSettings si les feuilles ne doivent pas toutes avoir le même paramétrage.

PrintPage

Cet évènement est utilisé pour positionner le code d'impression. Afin de pouvoir gérer celui-ci, on utilise l'argument PrintPageEventArgs qui contient, entre autre, les paramètres suivants :

- **Cancel** (bool) → Permet d'annuler le travail d'impression
- **HasMorePages** (bool) → Vrai s'il reste des pages à imprimer après la page en cours
- **MarginBounds** (Rectangle) → Rectangle qui représente la zone à l'intérieur des marges
- **PageSettings** (PageSettings) → Paramètres de la page en cours d'impression. Tous les paramètres d'un travail sont stockés dans la propriété DefaultPageSettings du PrintDocument mais chaque page peut être gérée séparément par le biais de ce paramètre.

N'oublions pas non plus le paramètre Graphics sur lequel va porter le travail puisqu'une impression n'est jamais qu'un dessin sur une imprimante.

Comme un petit dessin vaut mieux qu'un long discours, regardons ensemble un exemple.

Impression d'une grille de données

Dans cet exemple, nous allons imprimer une grille de données. Je vais utiliser le code de remplissage suivant :

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim MaConn As New OleDbConnection("Data
Source=D:\User\BIBLIO.MDB;Provider=Microsoft.Jet.OLEDB.4.0")
    Dim MonAdapt As New OleDbDataAdapter("SELECT * From Publishers",
MaConn)
    Dim MaTable As New DataTable
    MonAdapt.Fill(MaTable)
    Me.DataGrid1.DataSource = MaTable
End Sub
```

Et je veux pouvoir imprimer une grille contenant quelques champs de ma visualisation écran. Sur mon formulaire, j'ajoute donc :

- Un contrôle PrintDocument
- Un contrôle PageSetupDialog
- Un contrôle PrintPreviewDialog

En fait, je ne vais pas travailler en me basant sur le Datagrid mais uniquement sur la Table. Il est parfois mieux de se baser sur la grille si on veut prendre en compte la mise en forme utilisateur, tout dépend de ce que l'on veut obtenir.

N'oubliez pas que les deux boites de dialogues doivent avoir le PrintDocument1 défini dans leur propriété Document.

Il existe toujours plusieurs façons d'envisager un gestionnaire d'impression. Par habitude du VB6, j'utilise toujours une structure. Il serait cependant plus adapté maintenant de passer par une classe. Cependant je vais garder mon approche traditionnelle puisqu'elle est parfaitement valide.

Mon code va commencer comme suit :

```
    Private Enum Champ
        PubId
        Name
        Company_Name
        Address
        City
        State
        Zip
        Telephone
        Fax
```

```
        Comments
    End Enum

    Private Structure Tableau
        Dim NbRow As Int32
        Dim RowPerPage As Int32
        Dim Hauteur As Single
        Dim Largeur As Single
        Dim TabLarg() As Single
    End Structure


    Private TabImpr As New Tableau
    Private NombrePage As Int32 = 0
    Private PageNum As Int32 = 0

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Dim MaConn As New OleDbConnection("Data
Source=D:\tutoriel\BIBLIO.MDB;Provider=Microsoft.Jet.OLEDB.4.0")
        Dim MonAdapt As New OleDbDataAdapter("SELECT * From Publishers",
MaConn)
        Dim MaTable As New DataTable
        MonAdapt.Fill(MaTable)
        Me.DataGrid1.DataSource = MaTable
    End Sub
```

La première énumération me permettra d'atteindre la colonne de mon choix en utilisant un index mais avec un code plus lisible. En effet, on peut toujours appeler une colonne par son nom ou par son index. L'appel par le nom est plus lent mais plus explicite, l'énumération me permet d'avoir le meilleur des deux mondes.

Ma structure va contenir les éléments dont j'ai besoin pour l'impression, telle que :

- NbRow → Nombre de ligne de la table
- RowPerPage → Nombre de ligne par page
- Hauteur → Hauteur d'une ligne (1/100 inch)
- Largeur → Largeur de toutes les colonnes réunies
- TabLarg() → Tableau des positions des colonnes (1/100 inch – cumulé)

Je déclare enfin deux variables globales pour gérer le nombre de page et la page en cours

Je vais donc ensuite coder le bouton qui gère l'enchaînement de l'impression, en l'occurrence les deux boites de dialogue. Son code sera :

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    'paramétrage
    With Me.PrintDocument1.DefaultPageSettings
        .Landscape = True
        .Color = False
        .Margins = New Printing.Margins(ConvTo(10), ConvTo(10),
ConvTo(10), ConvTo(10))
    End With
    With Me.PageSetupDialog1
        .AllowMargins = True
        .AllowOrientation = False
        .AllowPaper = True
        .AllowPrinter = True
        If System.Globalization.RegionInfo.CurrentRegion.IsMetric Then
            .MinMargins = New Printing.Margins(100, 100, 100, 100)
```

```
        Else
            .MinMargins = New Printing.Margins(ConvTo(10), ConvTo(10),
ConvTo(10), ConvTo(10))
        End If
    End With
    If Me.PageSetupDialog1.ShowDialog(Me) = DialogResult.Cancel Then
        If MessageBox.Show("Voulez vous abandonnez l'impression",
"Annulation", MessageBoxButtons.OKCancel, MessageBoxIcon.Question,
MessageBoxDefaultButton.Button1, MessageBoxOptions.DefaultDesktopOnly) =
DialogResult.OK Then
            Exit Sub
        End If
    End If
    ReDim TabImpr.TabLarg(4)
    Dim MaConn As New OleDbConnection("Data
Source=D:\tutoriel\BIBLIO.MDB;Provider=Microsoft.Jet.OLEDB.4.0")
    Dim MaCommand As New OleDbCommand("SELECT Max(Len([Name])) AS lNom,
Max(Len([Address])) AS lAdresse, Max(Len([City])) AS lVille,
Max(Len([State])) AS lEtat FROM Publishers", MaConn)
    MaConn.Open()
    'Récupération des longueurs de chaines
    Dim dtrTaille As OleDbDataReader =
MaCommand.ExecuteReader(CommandBehavior.SingleRow)
    dtrTaille.Read()
    For cmpt As Int32 = 0 To 3
        TabImpr.TabLarg(cmpt) = dtrTaille.GetInt32(cmpt)
    Next
    dtrTaille.Close()
    'récupération du nombre de lignes
    MaCommand.CommandText = "SELECT COUNT(PubID) FROM Publishers"
    TabImpr.NbRow = CInt(MaCommand.ExecuteScalar)
    MaConn.Close()
    Me.PrintPreviewDialog1.ShowDialog()

End Sub
```

La première partie de ce code me sert uniquement à paramétrer le document et la boite de dialogue PageSetup. Notez que je vérifie les paramètres de CurrentRegion pour les marges car il y a un bug dans la boite de dialogue si vous êtes en système métrique (et oui, je sais….).

Dans la seconde partie, je vais commencer à remplir ma structure.

En effet, pour gérer correctement mon encadrement je dois connaître la taille maximale des chaînes de chaque champ et le nombre de ligne. Je pourrais récupérer ces informations de ma table mais, dans ce cas, je passe par des commandes. Cela nous permet de voir rapidement l'appel de commande en VB.NET.

Notez bien que pour l'instant, ma structure TabImpr.TabLarg() stocke des longueurs de chaîne.

Tout le reste va se passer au niveau de l'évènement principal de l'impression PrintPage.

```
Private Sub PrintDocument1_PrintPage(ByVal sender As Object, ByVal e As
System.Drawing.Printing.PrintPageEventArgs) Handles
PrintDocument1.PrintPage

    Dim PHauteur As Single = CSng(e.MarginBounds.Height)
    Dim ImprFont As New Font("Arial", 10)
    Dim ImprBrush As New SolidBrush(Color.Black)
    Dim ImprFormat As New StringFormat
    ImprFormat.Alignment = StringAlignment.Center
    ImprFormat.LineAlignment = StringAlignment.Center
```

```
    Dim gr As Graphics = e.Graphics
    Dim XBase As Int32 = CInt(e.MarginBounds.Left)
    Dim YBase As Int32 = CInt(e.MarginBounds.Top)
    'Fin du remplissage de la structure
    If NombrePage = 0 Then
        With gr
            For cmpt As Int32 = 0 To 3
                TabImpr.TabLarg(cmpt) = CInt(.MeasureString(New
String(CType("X", Char), CInt(TabImpr.TabLarg(cmpt))), New Font("Arial",
10)).ToPointF.X)
                TabImpr.Largeur = TabImpr.Largeur + TabImpr.TabLarg(cmpt)
                If cmpt > 0 Then TabImpr.TabLarg(cmpt) =
TabImpr.TabLarg(cmpt) + TabImpr.TabLarg(cmpt - 1)
            Next
            TabImpr.Hauteur = CInt(.MeasureString("X", New Font("Arial",
10)).Height)
            TabImpr.RowPerPage = CInt(PHauteur / TabImpr.Hauteur) - 1
            NombrePage = CInt(IIf(TabImpr.NbRow Mod TabImpr.RowPerPage =
0, TabImpr.NbRow \ TabImpr.RowPerPage, TabImpr.NbRow \ TabImpr.RowPerPage
+ 1))
        End With
    End If
    'Impression des titres
    gr.DrawString("Editeur", ImprFont, ImprBrush, New RectangleF(XBase,
YBase, TabImpr.TabLarg(0), TabImpr.Hauteur), ImprFormat)
    gr.DrawRectangle(New Pen(Color.Black, 1), XBase, YBase,
TabImpr.TabLarg(0), TabImpr.Hauteur)
    gr.DrawString("Adresse", ImprFont, ImprBrush, New RectangleF(XBase +
TabImpr.TabLarg(0), YBase, TabImpr.TabLarg(1) - TabImpr.TabLarg(0),
TabImpr.Hauteur), ImprFormat)
    gr.DrawRectangle(New Pen(Color.Black, 1), XBase + TabImpr.TabLarg(0),
YBase, TabImpr.TabLarg(1) - TabImpr.TabLarg(0), TabImpr.Hauteur)
    gr.DrawString("Ville", ImprFont, ImprBrush, New RectangleF(XBase +
TabImpr.TabLarg(1), YBase, TabImpr.TabLarg(2) - TabImpr.TabLarg(1),
TabImpr.Hauteur), ImprFormat)
    gr.DrawRectangle(New Pen(Color.Black, 1), XBase + TabImpr.TabLarg(1),
YBase, TabImpr.TabLarg(2) - TabImpr.TabLarg(1), TabImpr.Hauteur)
    gr.DrawString("Etat", ImprFont, ImprBrush, New RectangleF(XBase +
TabImpr.TabLarg(2), YBase, TabImpr.TabLarg(3) - TabImpr.TabLarg(2),
TabImpr.Hauteur), ImprFormat)
    gr.DrawRectangle(New Pen(Color.Black, 1), XBase + TabImpr.TabLarg(2),
YBase, TabImpr.TabLarg(3) - TabImpr.TabLarg(2), TabImpr.Hauteur)
    'impression des données
    Dim MaTable As DataTable = CType(Me.DataGrid1.DataSource, DataTable)
    Dim PosLigne As Int32 = (PageNum * TabImpr.RowPerPage)
    Try
        For cmptRow As Int32 = PosLigne To PosLigne + TabImpr.RowPerPage
- 1

gr.DrawString(MaTable.Rows(cmptRow).Item(Champ.Name).ToString, ImprFont,
ImprBrush, New RectangleF(XBase, YBase + ((cmptRow - PosLigne + 1) *
TabImpr.Hauteur), TabImpr.TabLarg(0), TabImpr.Hauteur), ImprFormat)
            gr.DrawRectangle(New Pen(Color.Black, 1), XBase, YBase +
((cmptRow - PosLigne + 1) * TabImpr.Hauteur), TabImpr.TabLarg(0),
TabImpr.Hauteur)
```

```
gr.DrawString(MaTable.Rows(cmptRow).Item(Champ.Address).ToString,
ImprFont, ImprBrush, New RectangleF(XBase + TabImpr.TabLarg(0), YBase +
((cmptRow - PosLigne + 1) * TabImpr.Hauteur), TabImpr.TabLarg(1) -
TabImpr.TabLarg(0), TabImpr.Hauteur), ImprFormat)
            gr.DrawRectangle(New Pen(Color.Black, 1), XBase +
TabImpr.TabLarg(0), YBase + ((cmptRow - PosLigne + 1) * TabImpr.Hauteur),
TabImpr.TabLarg(1) - TabImpr.TabLarg(0), TabImpr.Hauteur)

gr.DrawString(MaTable.Rows(cmptRow).Item(Champ.City).ToString, ImprFont,
ImprBrush, New RectangleF(XBase + TabImpr.TabLarg(1), YBase + ((cmptRow -
PosLigne + 1) * TabImpr.Hauteur), TabImpr.TabLarg(2) -
TabImpr.TabLarg(1), TabImpr.Hauteur), ImprFormat)
            gr.DrawRectangle(New Pen(Color.Black, 1), XBase +
TabImpr.TabLarg(1), YBase + ((cmptRow - PosLigne + 1) * TabImpr.Hauteur),
TabImpr.TabLarg(2) - TabImpr.TabLarg(1), TabImpr.Hauteur)

gr.DrawString(MaTable.Rows(cmptRow).Item(Champ.State).ToString, ImprFont,
ImprBrush, New RectangleF(XBase + TabImpr.TabLarg(2), YBase + ((cmptRow -
PosLigne + 1) * TabImpr.Hauteur), TabImpr.TabLarg(3) -
TabImpr.TabLarg(2), TabImpr.Hauteur), ImprFormat)
            gr.DrawRectangle(New Pen(Color.Black, 1), XBase +
TabImpr.TabLarg(2), YBase + ((cmptRow - PosLigne + 1) * TabImpr.Hauteur),
TabImpr.TabLarg(3) - TabImpr.TabLarg(2), TabImpr.Hauteur)
        Next
    Catch ex As Exception

    Finally
        gr.Dispose()
        PageNum += 1
        If PageNum < NombrePage Then e.HasMorePages = True
    End Try

End Sub
```

Si le code n'est pas long, il n'est pas évident non plus. Regardons cela de plus près. Je crée d'abord des objets Font, Brush, StringFormat dont je vais avoir besoin pour mes appels de méthode DrawString et DrawRectangle.

Je finis ensuite de remplir ma structure. Pour l'instant, j'ai des tailles de chaînes, mais il me faut des positions géométriques, en 1/100^{éme} de pouce, pour pouvoir imprimer correctement. Pour faire cette conversion, je vais invoquer la méthode MeasureString de l'objet graphique, tel que :

```
TabImpr.TabLarg(cmpt) = CInt(.MeasureString(New String(CType("X", Char),
CInt(TabImpr.TabLarg(cmpt))), New Font("Arial", 10)).ToPointF.X)
```

Ceci revient à dire :

- crée une chaîne de n caractères "X"
- considère une police Arial 10
- renvoie-moi la largeur en centième de pouce.

Je récupère aussi la hauteur de la même façon.

Vous noterez que je cumule mes positions dans mon tableau TabImpr.TabLarg(). Autrement dit on peut le lire tel que

TabImpr.TabLarg(0) = Largeur de la première colonne
TabImpr.TabLarg(1) = Position droite de la deuxième colonne
TabImpr.TabLarg(2)= Position droite de la troisième colonne
Etc…

Ceci implique que pour obtenir une largeur d'une colonne autre que la première, je devrai faire la soustraction entre sa position droite et celle de la colonne précédente, que pour obtenir sa position gauche je n'aurais qu'à lire la position droite de l'élément précédent, etc…

Enfin je calcule le nombre de ligne par page et le nombre de page.

Ensuite nous rentrons dans le travail d'impression proprement dit. A chaque page, j'imprimerai la ligne de titre. Pour chaque colonne, je trace le texte et le cadre. Les deux méthodes, DrawString et DrawRectangle acceptent de nombreuses surcharges. Pour ma part j'utilise :

Graphics.DrawString(String, Font, Brush, RectangleF, StringFormat)

Et

Graphics.DrawRectangle(Pen, Single, Single, Single, Single)

Je n'ai plus ensuite qu'à écrire mes lignes et mes rectangles sur autant de page que nécessaire.

Notez que j'utilise un bloc Try…Catch…Finally pour l'impression des données.

En effet, il n'y a quasiment aucune chance que le nombre de ligne tombe juste à la dernière page. Ce qui fait que ma boucle risque d'invoquer des appels sur ma table avec un numéro de ligne qui n'existe pas. C'est pour éviter ce problème que j'utilise le bloc.

Nous n'avons fais ici qu'un rapide survol de l'impression, j'essayerais d'écrire un cours plus complet sur le sujet à l'occasion.

Conclusion

Voilà, nous en avons fini avec cette troisième partie. Vous possédez maintenant suffisamment d'éléments pour migrer vos connaissances vers VB.NET. Il y a un temps d'adaptation nécessaire dans cette mise à niveau, mais vous verrez qu'il est beaucoup plus rapide que vous auriez pu le penser.

N'hésitez pas à venir poser vos questions sur les forums de developpez.com :
Général Dotnet
WindowsForms & Applications Windows
Webforms & Développement Web