Haute Ecole de la Ville de Liège

Catégorie Technique

Les applications WPF sous C#

A l'usage des étudiants en 2ème Informatique et Systèmes (Technologies de l'Informatique)

Année académique 2016-2017
Patrick Alexandre

0. Introduction

0.1. XAML – WPF – C#

WPF (Windows Presentation Foundation) est une couche graphique du Framework .NET qui offre beaucoup plus de possibilités que les WinForms traditionnelles. WPF est entièrement vectoriel et permet ainsi d'adapter une fenêtre en fonction des résolutions et des redimensionnements sans pixellisation.

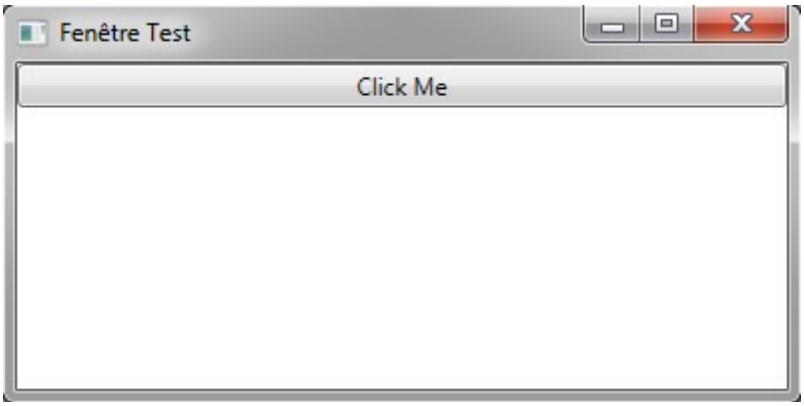
WPF permet de séparer facilement le design et la programmation. L'interface utilisateur est développée par l'intermédiaire de XAML (eXtensible Application Markup Language), langage utilisant des balises comme HTML, XML, … et, dans le cadre de WPF, enregistré sous la forme de fichiers texte d'extension xaml. La programmation en tant que telle est prise en charge par tout langage supporté par .Net (C#, Visual Basic, C++, …).

Le langage de programmation utilisé peut faire référence aux contrôles XAML déposés sur l'interface utilisateurs tout comme la fenêtre créée peut communiquer avec les données manipulées par le code source où les langages compatibles avec le Framework .Net .sont utilisables. Pour notre part, nous reterons fidèles à C#.

Sans entrer dans les détails, nous pourrions écrire

```
<StackPanel>
 <Button Content="Click Me"/>
</StackPanel>
```

et, ainsi, définir un bouton dans un contrôle *container*. Notons que cette balise sera, par la suite, interprétée lors de l'exécution pour représenter effectivement un bouton à l'écran.



Lors de la compilation du projet WPF, les fichiers xaml sont également compilés, compressés sous la forme BAML (Binary Application Markup Language) et associés au projet sous la forme de ressource.

0.2. La syntaxe en bref

Le principe de ce qui suit est de présenter les grands principes associés à XAML sans entrer dans les détails des balises utilisées. Ces mêmes détails seront abordés par la suite.

0.2.1. Les objets XAML

Tout élément XAML est en fait une instance d'un type défini dans les assemblies utilisées par le langage de programmation.

Tout élément débute par < et se termine par > ou /> selon que des attributs sont intégrés ou non. Directement après <, nous retrouvons le nom du type de balise utilisée que nous répétons devant />. L'exemple précédent illustre ces règles de déclaration. Pour rappel, nous avons écrit.

```
<StackPanel>
 <Button Content="Click Me"/>
</StackPanel>
```

Nous apprenons ainsi que `StackPanel` et `Button` sont deux classe mises à disposition du langage C#.

0.2.2. Définir des attributs

On peut personnaliser les contrôles déposés sur la fenêtre par le biais des propriétés. Pour ce faire, il suffit de renseigner l'option suivi du signe = puis de la valeur souhaitée encadrée par des guillemets.

Par exemple, nous pouvons définir un bouton de fond bleu et dont le texte est en rouge par l'instruction XAML suivante.

```
<Button Background="Blue" Foreground="Red" Content="This is a button"/>
```

Nous venons de voir ce que nous pouvons considérer comme étant un raccourci de définition de propriétés. En effet, il est possible de renseigner les propriétés comme étant des sous-éléments du contrôle à personnaliser.

Nous pouvons écrire le code suivant de manière équivalente à ce qui précède, la balise du contrôle étant évidemment fermée différemment.

```
<Button>
 <Button.Background>
  <SolidColorBrush Color="Blue"/>
 </Button.Background>
 <Button.Foreground>
  <SolidColorBrush Color="Red"/>
 </Button.Foreground>
 <Button.Content>
  This is a button
 </Button.Content>
</Button>
```

Partant de la définition précédente, il est possible de renseigner une liste de valeurs pour une propriété particulière. Ainsi, le code suivant permet de colorer le bouton, de la gauche vers la droite, en rouge, jaune et bleu, les couleurs étant mélangées sur les zones d'intersection. Nous écrivons

```
<Button>
 <Button.Background>
  <LinearGradientBrush>
   <LinearGradientBrush.GradientStops>
    <GradientStop Offset="0.0" Color="Red" />
    <GradientStop Offset="1.0" Color="Blue" />
    <GradientStop Offset="0.5" Color="Yellow" />
   </LinearGradientBrush.GradientStops>
  </LinearGradientBrush>
 </Button.Background>
 <Button.Foreground>
  <SolidColorBrush Color="Red"/>
 </Button.Foreground>
 <Button.Content>
  This is a button
 </Button.Content>
</Button>
```

Les balises `LinearGradientBrush.GradientStops` peuvent être supprimées, l'association se faisant automatiquement selon le contexte.

```
<LinearGradientBrush>
 <GradientStop Offset="0.0" Color="Red" />
 <GradientStop Offset="1.0" Color="Blue" />
 <GradientStop Offset="0.5" Color="Yellow" />
</LinearGradientBrush>
```

Le résultat est le suivant.

Il est également possible d'utiliser un contrôle comme propriété et l'indication de cette propriété devient alors superflue.

Par exemple, le code explicite suivant qui place un `TextBox` dans un cadre de couleur `Cyan` d'épaisseur `10` pixels

```
<Border BorderBrush="Cyan" BorderThickness="10">
 <Border.Child>
  <TextBox Height="30" Width="200"/>
 </Border.Child>
</Border>
```

peut être remplacé par

```
<Border BorderBrush="Aqua" BorderThickness="10">
 <TextBox Height="30" Width="200"/>
</Border>
```

Notons qu'il ne faut pas séparer les propriétés renseignées. C'est ainsi que l'écriture suivante est reconnue par le compilateur

```
<Button>
 Je suis un bouton
 <Button.Background>Green</Button.Background>
</Button>
```

contrairement à celle-ci.

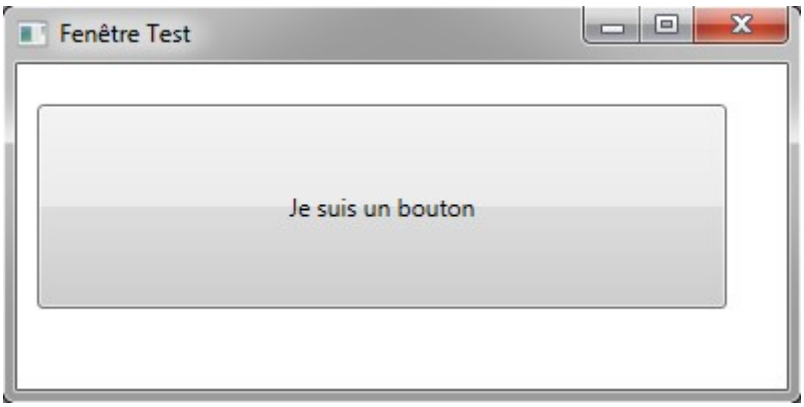
```
<Button>
 Je suis
 <Button.Background>Green</Button.Background>
 un bouton
</Button>
```

Nous verrons qu'il est possible de définir des convertisseurs de valeurs mais cetains sont livrés par XAML. Par exemple, il est possible de définir les marges (extérieures) d'un élément. Normalement, ces marges sont spécifiées distinctement pour la gauche, la supérieure, la droite et l'inférieure comme suit.

```
<Button Content="Je suis un bouton">
 <Button.Margin>
  <Thickness Left="10" Top="20" Right="30" Bottom="40" />
 </Button.Margin>
</Button>
```

Cette écriture peut être synthétisée sous la forme suivante.

```
<Button Content="Je suis un bouton" Margin="10,20,30,40" />
```



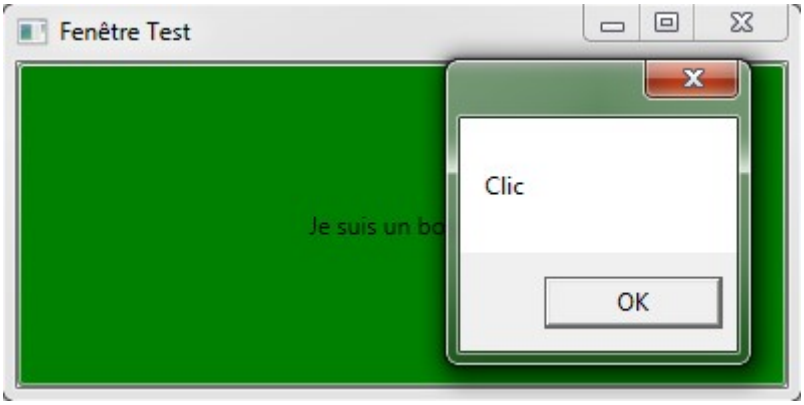
0.2.3. La gestion des évenements

Evidemment, il est possible de définir des événements associés à des contrôles XAML. Il suffit d'utiliser la propriété adéquate comme le clic sur un bouton comme dans le code suivant.

```
<Button Click="Button_Click">
 Je suis un bouton
 <Button.Background>Green</Button.Background>
</Button>
```

Nous retrouvons dans le code associé à la fenêtre la procédure `Button_Click` que nous pouvons simplement adapter de la manière suivante.

```
private void Button_Click(object sender, RoutedEventArgs e)
{ MessageBox.Show("Clic"); }
```



Ces événements peuvent prendre une forme plus complexe. En effet, il est possible de *router* les événements. Cela signifie que l'événement s'applique à plusieurs contrôles qui sont unis selon une arborescence.

Le code suivant présente un `StackPanel` composé de trois boutons dont le premier est intégré dans un second `StackPanel`.

Il n'y a rien de particulier à signaler si ce n'est que nous sommes confrontés à un événement attaché car les contrôles `StackPanel` ne disposent pas d'événement `Click` et nous contournons ce manque en attachant l'événement `Click` défini dans `Button`.

Il en est de même pour les propriétés.

```
<StackPanel Orientation="Horizontal" Button.Click="CommonClickHandler">
 <Button Name="YesButton" Width="Auto" >
  <StackPanel>
   <Image Source="image.png" Width="25"/>
   <Label>Yes</Label>
  </StackPanel>
 </Button>
 <Button Name="NoButton" Width="Auto" >No</Button>
 <Button Name="CancelButton" Width="Auto" >Cancel</Button>
</StackPanel>
```

Le code behind associé peut se mettre sous la forme suivante.

```
private void CommonClickHandler(object sender, RoutedEventArgs e)
{FrameworkElement feSource = e.Source as FrameworkElement;
 switch (feSource.Name)
 {case "YesButton":
   MessageBox.Show("YesButton");
   break;
  case "NoButton":
   MessageBox.Show("NoButton");
   break;
  case "CancelButton":
   MessageBox.Show("CancelButton");
   break;
 }
}
```

Alors, l'événement défini dans le `StackPanel` prend en charge la gestion du clic effectué sur n'importe quel bouton ainsi que sur l'image associée au bouton `YesButton`.

Pour terminer ce paragraphe sur les événements, signalons que tout contrôle XAML dont le nom est spécifié est accessible dans le code behind.

0.2.4. Les éléments racines et les espaces de noms

Un fichier XAML doit avoir un élément racine pour être valide. Le cas typique est de définir un fichier xaml associé à la création d'un formulaire Windows dans le cadre d'une application WPF. Nous pouvons ainsi retrouver, par défaut, un élément comme racine du fichier xaml.

```
<Window x:Class="SandBox.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="1000">
 <Grid>
 </Grid>
</Window>
```

Notons qu'un autre type de racine est également créé dans ce cadre, à savoir un élément de type `Application` dans le fichier App.xaml et se présente comme suit.

```
<Application
  x:Class="SandBox.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
 <Application.Resources>
 </Application.Resources>
</Application>
```

Parlons maintenant des attributs `xmlns` et `xmlns:x` que nous retrouvons dans ces éléments racines. Ils permettent de spécifier la référence utilisée pour définir la syntaxe XAML tout comme en HTML avec l'attribut `DOCTYPE`.

Notons également la présence, dans ces définitions d'éléments racines, du préfixe `x:`. Nous pouvons en fait le rencontrer habituellement sous les formes suivantes.

- `x:Key` définit une clé unique pour chaque ressource placée dans un dictionnaire (`ResourceDictionary`).
- `x:Class` renseigne la classe C# associée, en code behind, au fichier xaml.
- `x:Name` permet de définir un synonyme à un élément issu du code et propre au formulaire.
- `x:Static` rend disponible une référence qui retourne une valeur statique non compatible avec une propriété XAML.
- `x:Type` associe une référence à un type.

0.3. Générer du XAML en C#

Les contrôles disponibles en XAML existant sous la forme de classes en C#, il est possible de manipuler des contrôles XAML depuis le code behind et, par conséquent, on peut aussi les créer.

Penchons-nous sur le code XAML suivant.

```
<Grid Name="MaGrille">
 <Button Content="Bouton" Click="OnButtonClick" Width="100" Height="50">
  <Button.Background>
   <LinearGradientBrush>
    <GradientStop Color="Yellow" Offset="0" />
    <GradientStop Color="Green" Offset="1" />
   </LinearGradientBrush>
  </Button.Background>
 </Button>
</Grid>
```

L'équivalent en C#, pour créer le même bouton dans la grille `MaGrille` déjà existant peut prendre la forme suivante.

```
Button b = new Button();
b.Content = "Click Me";
b.Click += this.Button_Click;
LinearGradientBrush lgb = new LinearGradientBrush();
lgb.GradientStops.Add(new GradientStop(Colors.Yellow, 0));
lgb.GradientStops.Add(new GradientStop(Colors.Green, 1));
b.Background = lgb;
b.Width = 100;
b.Height = 50;
MaGrille.Children.Add(b);
```

1. Le layout

1.1. Introduction

Un layout permet de disposer les contrôles sur l'interface utilisateur.

C'est une partie non négligeable du développement d'une application car l'ergonomie de cette dernière jouera un grand rôle dans son succès … ou son échec.

Notons que la séparation entre le design et le code permet de séparer les métiers de designers et de développeurs. Néanmoins, les développeurs doivent également avoir une idée de ce qui est possible tout comme les designers doivent connaître les limites de la programmation.

Nous allons, dans ce chapitre, aborder les différentes possibilités relatives au placement sur la fenêtre des composants XAML. Les composants que nous allons présenter peuvent être associés à des containers qui permettent d'organiser, sur le plan visuel, la page.

1.2. Le Canvas

Le composant `Canvas` permet de placer les contrôles XAML selon les coordonnées X et Y.

Par conséquent, en cas de redimensionnement, les coordonnées étant définies en dur, on perd tout l'aspect *dynamique* de WPF. Ce composant ne répond donc pas tellement au but premier poursuivi par WPF mais, dans certains cas, il est néanmoins incontournable.

Pour se positionner dans un composant de type `Canvas`, nous disposons des propriétés suivantes qui ne nécessitent guère de commentaires sauf qu'il s'agit de propriétés attachées.

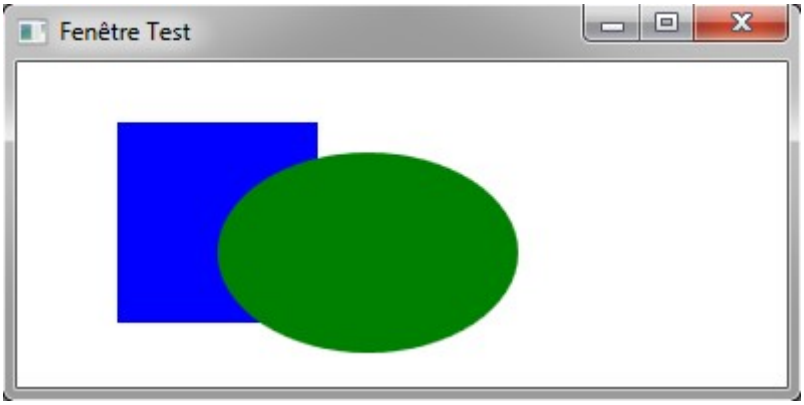
- `Canvas.Left`,
- `Canvas.Right`,
- `Canvas.Top`,
- `Canvas.Bottom`.

Par exemple, nous pouvons placer une image positionnée à 50 à droite du bord gauche et à 10 du bord inférieur de la manière suivante, l'image s'adaptant aux dimensions imposées.

```
<Canvas Background="Silver">
 <Image Source="image.png" Width="90" Height="90" Stretch="Fill"
        Canvas.Left="50" Canvas.Bottom="10"/>
</Canvas>
```

Signalons, pour terminer la présentation de ce composant, qu'il est possible de définir un ordre d'affichage par l'intermédiaire de la propriété `Canvas.ZIndex`. Ainsi, si nous nous laissons convaincre que les contrôles XAML suivants permettent de représenter à l'écran un rectangle (carré dans ce cas) et une ellipse, l'ellipse sera mise en avant plant via une valeur plus élevée pour l'attribut `Canvas.ZIndex` (0 par défaut).

```
<Canvas>
 <Rectangle Fill="Blue" Width="100" Height="100"
            Canvas.Left="50" Canvas.Top="30"/>
 <Ellipse Fill="Green" Width="150" Height="100"
          Canvas.ZIndex="1" Canvas.Left="100" Canvas.Top="45"/>
</Canvas>
```

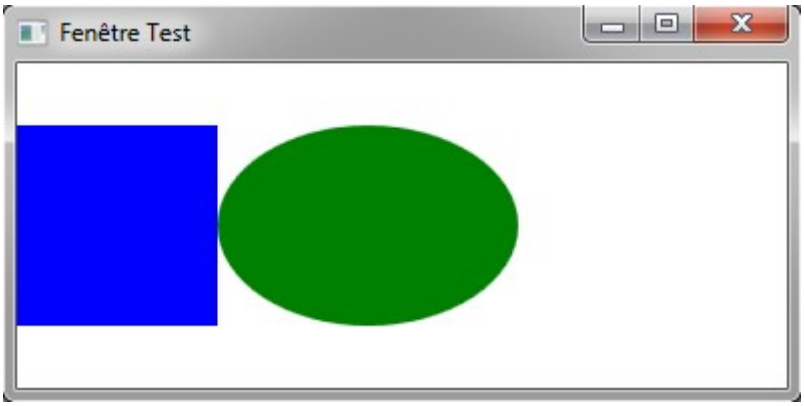


1.3. Le `StackPanel`

Ce composant permet d'aligner les contrôles qui le composent l'un à la suite de l'autre dans le sens vertical (par défaut) ou horizontal. La propriété `Orientation` permet de changer l'orientation d'affichage que l'on souhaite donner à ce composant.

Ainsi, pour afficher un rectangle placé à gauche du composant `StackPanel` et une ellipse placée directement à la droite de ce même rectangle, le code suivant fait l'affaire.

```
<StackPanel Margin="0" Orientation="Horizontal">
 <Rectangle Fill="Blue" Width="100" Height="100"
            Canvas.Left="50" Canvas.Top="100"/>
 <Ellipse Fill="Green" Width="150" Height="100"
          Canvas.Left="100" Canvas.Top="125"/>
</StackPanel>
```



Ce composant dispose d'une *version* particulière, `VirtualizingStackPanel`, qui consiste à traiter le cas où les éléments qui y seront associés sont en nombre (plus) important (que les éléments qui y seront affichés).

Par défaut, ce composant est implémenté sur les `ListBox`, les `ListView` et les `TreeView`. Ces derniers peuvent utiliser la virtualisation annoncée par l'intermédiaire de leur propriété attachée `VirtualizingStackPanel.IsVirtualizing`.

Lorsque nous souhaitons afficher une partie d'enregistrements tirés d'une base de données, il est possible que ces enregistrements soient en nombre important. Par conséquent, le temps de chargement devient non négligeable. Les programmeurs peuvent alors se limiter à charger les enregistrements qui sont destinés à être affichés directement, l'application devenant ainsi plus rapidement opérationnelle. Evidemment, il faut peser le pour et le contre entre le rapatriement en une fois d'un nombre important de données et la création des données à afficher ainsi que la destruction de celles qui ne doivent plus l'être …

La propriété attachée `VirtualizingStackPanel.VirtualizationMode` est la seule sur laquelle nous devons nous attarder. Si elle définie à `Recycling`, le composant réutilise les coteneurs d'éléments au lien d'en créer de nouveaux à chaque fois.

Par exemple, nous créons une `ListBox`.

```
<ListBox Name="ListBoxWithVirtualization" Height="100" Margin="5"
         VirtualizingStackPanel.IsVirtualizing="True"
         VirtualizingStackPanel.VirtualizationMode="Standard" />
```

Nous remplissons cette dernière par code dans le constructeur de la fenêtre qui prend la forme suivante.

```
public MainWindow()
{InitializeComponent();
 for (int i = 0; i < 100000; i++)
  ListBoxWithVirtualization.Items.Add("Item - " + i.ToString());
}
```

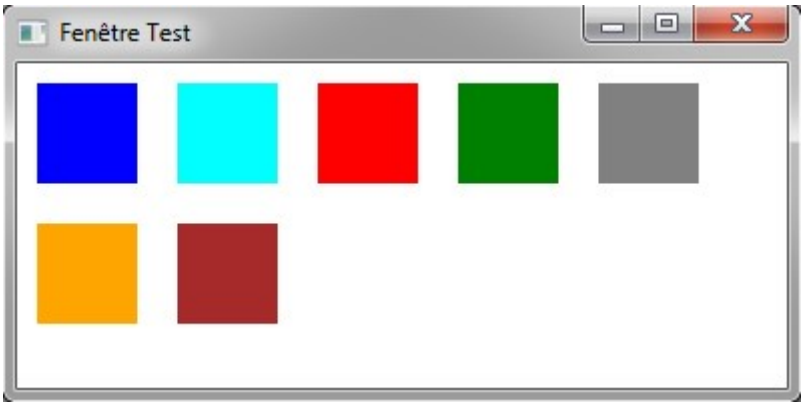
Si nous mettons `False`, nous constatons que les performances diminuent dramatiquement. Notons également que si nous ne renseignons rien, une virtualisation par défaut est de toute manière mise en place …

1.4. Le WrapPanel

Le `WrapPanel` est semblable au composant précédent si ce n'est qu'il décale les éléments en surnombre sur la droite (orientation verticale) ou vers le bas (orientation horizontale).

Nous pouvons considérer le code suivant, similaire à celui qui précède mais avec plus d'éléments visuels pour pouvoir constater ce décalage.

```
<WrapPanel Margin="0" Background="White" Orientation="Horizontal">
 <Rectangle Margin="10"
            Fill ="Blue" Width="50" Height="50"/>
 <Rectangle Margin="10"
            Fill ="Cyan" Width="50" Height="50"/>
 <Rectangle Margin="10"
            Fill ="Red" Width="50" Height="50"/>
 <Rectangle Margin="10"
            Fill ="Green" Width="50" Height="50"/>
 <Rectangle Margin="10"
            Fill ="Gray" Width="50" Height="50"/>
 <Rectangle Margin="10"
            Fill ="Orange" Width="50" Height="50"/>
 <Rectangle Margin="10"
            Fill ="Brown" Width="50" Height="50"/>
</WrapPanel>
```



1.5. Le DockPanel

Nous rencontrons ici un composant qui peut être considéré comme étant la base de toute application de type fenêtré. Il permet de diviser l'écran en zones occupant une partie de l'écran.

Par exemple, les menus d'application se trouvent au dessus de la fenêtre. Il suffit, pour ce faire, de définir un `DockPanel` dont la position est définie au dessus de la fenêtre via la propriété

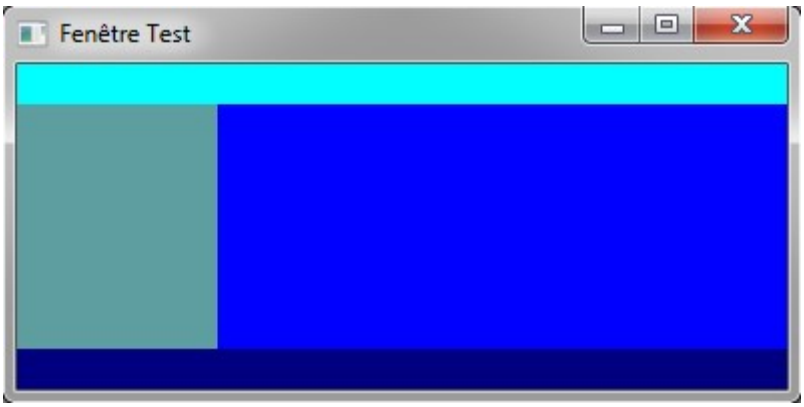
```
DockPanel.Dock="Top"
```

Au niveau des proprieties d'un tel composant, il faut également signaler la possibilité d'exiger que le dernier contrôle placé dans le composant occupe toute la place restante via

```
LastChildFill="True"
```

Nous allons, dans le code suivant, subdiviser l'écran en 4 zones. Deux sont horizontales. La première (`Cyan`) est située au dessus de la fenêtre symboliser le menu. La deuxième (`Navy`) est placée en dessous de la fenêtre et permet de symboliser une barre d'état. Ensuite viennent deux zones verticales dans le reste de la fenêtre. Cela revient à insérer un nouveau `DockPanel` destiné à contenir deux élements, l'un (`CadetBlue`) fixé à gauche et de largeur déterminée et l'autre (`Blue`) à droite. Pour terminer, signalons que le premier `DockPanel` demande que le dernier contrôle, à savoir le second `DockPanel`, occupe tout l'espace disponible via la propriété `LastChildFill` tandis que, dans le second `DockPanel` le remplissage se fait en fixant la largeur du premier rectangle et en spécifiant les propriétés `DockPanel.Dock`.

```
<DockPanel Width="Auto" Height="Auto" LastChildFill="True">
 <Rectangle Fill="Cyan" Height="20" DockPanel.Dock="Top"/>
 <Rectangle Fill="Navy" Height="20" DockPanel.Dock="Bottom"/>
 <DockPanel Width="Auto" Height="Auto">
  <Rectangle Fill="CadetBlue" Width="100" DockPanel.Dock="Left"/>
  <Rectangle Fill="Blue" DockPanel.Dock="Right"/>
 </DockPanel>
</DockPanel>
```



1.6. Le `Grid`

Ce composant agit comme l'ancienne balise HTML `<table>` avec la possibilité de définir lignes et colonnes formant un ensemble de cellules dont certaines peuvent être fusionnées.

La définition de lignes se fait via les instructions respectives suivantes

```
<Grid.RowDefinitions>
</Grid.RowDefinitions>
```

tandis que, pour les colonnes, nous avons

```
<Grid.ColumnDefinitions>
</Grid.ColumnDefinitions>
```

Comme contrôle enfant de ces composants, nous pouvons spécifier, par les propriétés insérées,

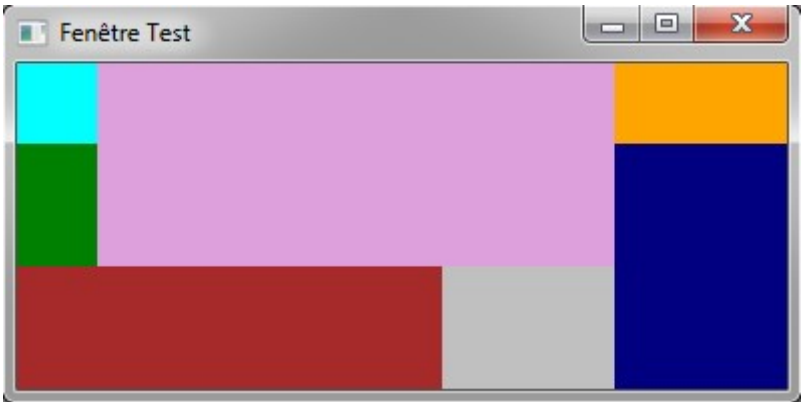
- les lignes par `<RowDefinition Height="40"/>`,
- les colonnes par `<ColumnDefinition Width="40"/>`.

En provenance de cette balise HTML, signalons le dimensionnement des lignes et colonnes qui peut se faire en *dur* comme ci-dessus en spécifiant explicitement une largeur ou de manière relative par l'intermédiaire de proportions obtenues via le symbole `*`.

A partir du moment où la grille est définie, les composants qui la peuplent font simplement référence à la ligne (`Grid.Row`) et à la colonne (`Grid.Column`) à laquelle ils *appartiennent* et le positionnement se fait alors automatiquement, 0 étant la valeur par défaut. Comme promis, il est également possible de spécifier le nombre de cellules (horizontalement (`Grid.ColumnSpan`) et/ou verticalement (`Grid.RowSpan`)) fusionnées pour prendre en charge le contrôle.

L'exemple qui suit s'appuie sur une grille de 3 lignes sur 4 colonnes remplie de rectangles définis aussi sur des cellules fusionnées.

```
<Grid Width="Auto" Height="Auto" >
 <Grid.RowDefinitions>
  <RowDefinition Height="40"/>
  <RowDefinition Height="*"/>
  <RowDefinition Height="*"/>
 </Grid.RowDefinitions>
 <Grid.ColumnDefinitions>
  <ColumnDefinition Width="40"/>
  <ColumnDefinition Width="2*"/>
  <ColumnDefinition Width="*"/>
  <ColumnDefinition Width="*"/>
 </Grid.ColumnDefinitions>
 <Rectangle Fill="Aqua"/>
 <Rectangle Fill="Plum"
            Grid.Column="1" Grid.ColumnSpan="2" Grid.RowSpan="2"/>
 <Rectangle Fill="Orange"
            Grid.Column="3"/>
 <Rectangle Fill="Green"
            Grid.Row="1"/>
 <Rectangle Fill="Navy"
            Grid.Row="1" Grid.Column="3" Grid.RowSpan="2"/>
 <Rectangle Fill="Brown"
            Grid.Row="2" Grid.ColumnSpan="2"/>
 <Rectangle Fill="Silver"
            Grid.Row="2" Grid.Column="2"/>
</Grid>
```

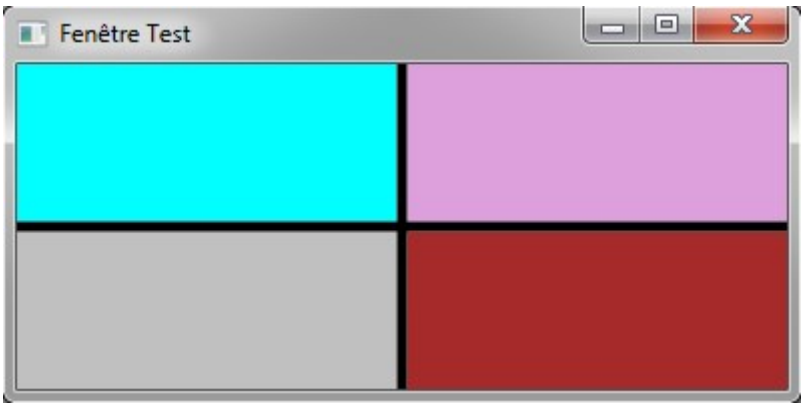


Nous pouvons également associer au composant `Grid` le composant `GridSplitter` qui lui est associé. Il permet de placer, dans une ligne ou dans une colonne réservée à cet effet, un séparateur qui donne la possibilité à l'utilisateur de redimensionner dynamiquement les cellules.

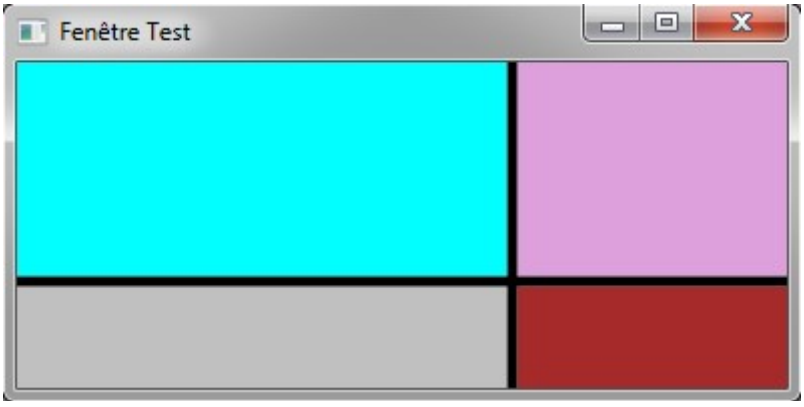
Par exemple, nous pouvons écrire le code suivant qui contient 4 cellules contenant chacune un rectangle et deux `GridSplitter` qui partagent horizontalement et verticalement la fenêtre.

```
<Grid Width="Auto" Height="Auto" >
 <Grid.RowDefinitions>
  <RowDefinition Height="*"/>
  <RowDefinition Height="Auto"/>
  <RowDefinition Height="*"/>
 </Grid.RowDefinitions>
 <Grid.ColumnDefinitions>
  <ColumnDefinition Width="*"/>
  <ColumnDefinition Width="Auto"/>
  <ColumnDefinition Width="*"/>
 </Grid.ColumnDefinitions>
 <Rectangle Fill="Aqua"/>
 <Rectangle Fill="Plum" Grid.Column="2" />
 <Rectangle Fill="Silver" Grid.Row="2"/>
 <Rectangle Fill="Brown" Grid.Row="2" Grid.Column="2" />
 <GridSplitter Grid.Row="1" Grid.ColumnSpan="3"
               HorizontalAlignment="Stretch" VerticalAlignment="Center"
               ShowsPreview="True" Background="Black" Height="5" />
 <GridSplitter Grid.Column="1" Grid.RowSpan="3"
               VerticalAlignment="Stretch" HorizontalAlignment="Center"
               ShowsPreview="True" Background="Black" Width="5" />
</Grid>
```

Le résultat est le suivant



et peut devenir, à l'exécution,



2. XAML versus code

Le fait de pouvoir effectuer la plupart des operations en XAML ou en C# entraîne évidemment la question du meilleur choix pour arriver au résultat final.

2.1. Opérations en XAML

Le fait que XAML permette de séparer le code de la vue ne doit pas quitter notre esprit et il est évidemment conseiller de laisser en XAML tout ce qui a trait à l'interface utilisateur comme c'est déjà le cas dans les applications fenêtrées classiques.

Il convient également de renseigner les fichiers qui servent de ressources à la fenêtre dans le code XAML.

Nous allons revenir en détail sur les ressources locales et sur les diverses associations possibles avec des objets définis par ailleurs (binding). Il est de mise de les définir dans le code XAML même si cette partie peut être déplacée vers le code C# si on souhaite vraiment rendre inaccessible toute référence au code behind au *designer*.

2.2. Opérations en C#

Classiquement, le code C# est destiné à stocker toute la gestion de la fenêtre et des contrôles qui en font partie. Cette gestion revient finalement à rédiger le code associée aux événements définis dans le code XAML.

Toutefois, certaines possibilités concernant l'animation sur des contrôles XAML peut être laissée au code XAML quoique, à nouveau, on préférera laisser les diverses définitions dans le code XAML et la manipulation de ces dernières dans le code C#.

Nous n'allons pas nous attarder sur la séparation entre les codes XAML et C# parce que, dans le cadre de ce cours, vu l'interaction avec les bases de données, nous allons nous pencher sur la modélisation MVVM (Model – View – View-Model).

2.3. Référencer en XAML

Un des problèmes essentiels est de pouvoir référencer les classes ou les contrôles utilisateurs définis ailleurs que dans le code XAML de la page en cours.

Nous allons examiner successivement les 4 cas qui peuvent se présenter à nous. Notons déjà que l'essentiel des manipulations se fait dans la partie déclarative de la fenêtre (balise `<Window >`) par l'intermédiaire de `xmlns`.

2.3.1. Utiliser un contrôle utilisateur de même *NameSpace* que la page

Nous supposons que le *NameSpace* de notre page soit `MonEspace`. Le contrôle utilisateur est supposé être `MonControle`.

Nous définissons un élément que l'habitude veut que l'on nomme `local`. Dans ce cas, la déclaration suivante convient parfaitement.

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 x:Class="MonEspace.MainWindow"
 xmlns:local="clr-namespace:MonEspace;assembly="
```

L'*assembly* ne devant pas être renseignée, l'option peut être ignorée comme suit.

```
xmlns:local="clr-namespace:MonEspace;"
```

Pour utiliser le contrôle, il suffit d'insérer là où nous souhaitons le voir apparaître, l'instruction XAML suivante.

```
<local:MonControle />
```

2.3.2. Utiliser une classe de même *NameSpace* que la page

La différence entre une classe et un contrôle utilisateur vient du fait que le contrôle fait partie de l'interface utilisateur et peut être assimilé à un contrôle *visuel*.

Une classe, par contre, doit être vue comme une ressource de la page. Nous supposons, dans ce qui suit, que la classe s'appelle `MaClasse`.

La déclaration de la référence à `MonEspace` au sein de la balise `<Window >` est identique au cas précédent. Pour ce qui est de l'utilisation de la classe en elle-même, en supposant que la classe soit référencée depuis la page, nous pouvons écrire le code suivant.

```
<Window

 <Window.Resources>
  <local:MaClasse x:Key="MaClasse1" Propriete="5"/>
 </Window.Resources>
```

En fait, nous définissons une instance de la classe `MaClasse` dans le code XAML qui sera prise en charge par l'objet `MaClasse1` à qui on assigne la propriété `Propriete` à `5`.

Si nous souhaitons utiliser cet objet `MaClasse1` instancié en XAML dans le code C#, nous pouvons, par exemple, placer le code suivant dans le constructeur de la page, après le chargement des contrôles.

```
public MainWindow()
 {InitializeComponent();
  MaClasse mc = FindResource("MaClasse1") as MaClasse;
  int temoin = mc.Propriete;
 }
```

La variable `temoin` récupère alors le contenu de `Propriete` stockée dans `MaClasse1`, à savoir `5`.

2.3.3. Utiliser un contrôle utilisateur d'un *NameSpace* autre que celui de la page

Supposons que le contrôle `AutreControle` soit défini dans un autre espace de noms baptisé, pour la bonne cause, `AutreEspace`. Nous pouvons également supposer que la librairie où se trouve la définition du contrôle soit `Librairie.dll`.

Comme on peut s'en douter à la lecture de 2.3, il nous suffit de renseigner l'espace de noms dans la définition de la référence par l'intermédiaire de l'option `xmlns`, la convention de baptême `local` n'étant plus vraiment d'actualité dans ce cas de figure. Cela se fait par l'intermédiaire de l'option `assembly=` comme suit

```
xmlns:ReferenceExterne="clr-namespace:AutreEspace;assembly=Librairie"
```

L'utilisation du contrôle se fait alors par le code XAML suivant.

```
<ReferenceExterne:AutreControle />
```

Notons que nous avons le choix sur le nom de baptême de notre référence XAML à l'objet et qu'il est de bon ton, pour s'y retrouver plus facilement dans le code, de baptiser cette référence du même nom que celui du *NameSpace*, comme suit

```
xmlns:AutreEspace="clr-namespace:AutreEspace;assembly=Librairie"
```

et

```
<AutreEspace:AutreControle />
```

2.3.4. Utiliser une classe d'un *NameSpace* autre que celui de la page

Conformément à ce que nous avons fait précédemment, nous supposons disposer d'une classe `AutreClasse` soit défini dans un autre espace de noms baptisé, pour la bonne cause, `AutreEspace`, le tout enregistré dans la librairie `Librairie.dll`.

Le référencement se fait comme suit

```
xmlns:AutreEspace="clr-namespace:AutreEspace;assembly=Librairie"
```

tandis que l'utilisation peut se mettre sous la forme suivante

```
<Window.Resources>
 <AutreEspace:AutreClasse x:Key="AutreClasse1"/>
</Window.Resources>
```

2.4. Markup extension

Une *markup extension* (*extension de balisage*), est une entité de programmation qui permet au processeur XAML de déléguer à une classe l'évaluation d'une propriété.

Une markup extension est identifiée dans le code XAML par l'utilisation d'accolades dans un attribut de propriété. Le contenu de ces accolades sera, en clair, évalué lors de l'exécution du code XAML en lien avec le code behind.

2.4.1. Première approche

Présentons le principe par un simple exemple où nous allons récupérer le résultat d'une variable d'environnement, `PATH` pour ne pas la citer.

Nous commençons par créer une classe `EnvironnementExtension` qui hérite de la classe de base `MarkupExtension`. La classe est annoncée par

```
[MarkupExtensionReturnType(typeof(string))]
```

qui signale que le type retourné par l'extension de balisage sera, dans ce cas, une chaîne de caractères. La méthode surchargée `ProvideValue` permet de calculer cette valeur qui va être retournée et qui possède un argument de type `IServiceProvider` sur lequel nous reviendrons plus tard.

Nous écrivons comme code de la classe.

```
[MarkupExtensionReturnType(typeof(string))]
public class EnvironnementExtension : MarkupExtension
{public string NomVariable { get; set; }
 public override object ProvideValue(IServiceProvider serviceProvider)
 {if (string.IsNullOrEmpty(NomVariable))
   throw new ArgumentException("La variable ne peut être vide");
  else
   return Environment.GetEnvironmentVariable(NomVariable);
 }
}
```

En fonction de ce que nous venons de voir précédemment, il nous suffit de référencer la classe `EnvironnementExtension` dans la balise `<Window >` puis d'utiliser cette même référence. Nous pouvons écrire le code XAML suivant.

```
<Window x:Class="SandBox.MainWindow"
        xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
        xmlns:local="clr-namespace:SandBox"
        xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
        Title="Fenêtre Test" Height="200" Width="400">
 <StackPanel>
  <TextBlock Text="Répertoires dans le PATH :"/>
  <TextBox Text="{local:EnvironnementExtension VariableName=PATH}"
           IsReadOnly="True" TextWrapping="Wrap" />
 </StackPanel>
</Window>
```

2.4.2. Ajout d'un constructeur

Contrairement au paragraphe précédent, nous allons définir, en plus du constructeur par défaut, un constructeur qui reçoit la variable d'environnement en argument. Nous complétons donc notre code par l'ajout des deux méthodes suivantes.

```
public EnvironnementExtension()
{ }
public EnvironnementExtension(string nomVariable)
{NomVariable = nomVariable;
}
```

Notons encore que la donnée membre `NomVariable` peut voir ses accesseurs implicites précédés d'une propriété qui permet de signaler que la propriété peut être initialisée via un argument du constructeur. Cette manière de faire est utile dans le cas où la classe est utilisée en mode design. Nous écrivons

```
[ConstructorArgument("NomVariable")]
public string NomVariable { get; set; }
```

Pour ce qui est de l'utilisation, nous pouvons écrire

```
<TextBox Text="{local:EnvironnementExtension PATH}"
         IsReadOnly="True" TextWrapping="Wrap" />
```

Les utilisations habituelles que l'on rencontre en tant que extensions de balisage sont les suivantes.

- `StaticResource` pour disposer d'un objet ou d'une valeur au moment de la création de la page,
- `DynamicResource` pour relier l'objet ou la valeur à tout moment,
- `Binding` permet d'associer un contrôle à une propriété d'un objet défini par ailleurs,
- `RelativeResource` renseigne une source de données pour les liens de type `Binding`, permettant ainsi d'adapter le contrôle au contexte dans lequel il se trouve à l'exécution,
- `x:Type` renseigne le type de l'objet,
- `x:static` pour associer une valeur *statique*,
- `x:Null` renvoie la valeur `null`,
- `x:Array` fournit la possibilité de manipuler des tableaux.

Développons quelque peu ces utilisations.

2.4.3. Ressource

Il est possible, en WPF, de définir des ressources. Cela revient habituellement à renseigner des objets ou valeurs souvent utilisés. Nous avons déjà vu que nous pouvions les utiliser dans les balises `<Window >` et `<Application >`. Il existe d'autres possibilités.

Les ressources définies sont associées à un identifiant par l'intermédiaire de l'attribut `x:Key` comme annoncé en 0.2.4. Par ce biais, un dictionnaire de ressources (`ResourceDictionary`) est élaboré.

Un exemple permet de mieux visualiser les différences entre ces deux concepts.

```
<Window x:Class="SandBox.MainWindow"
        xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
        xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
        Title="Fenêtre Test" Height="200" Width="400"
        Loaded="Window_Loaded">
 <Window.Resources>
  <SolidColorBrush Color="LightBlue" x:Key="RectangleFond" />
 </Window.Resources>
 <StackPanel Name="Panneau" VerticalAlignment="Center">
  <Rectangle Fill="{StaticResource RectangleFond}"
             Height="40" Width="100"/>
  <Rectangle Fill="{DynamicResource RectangleFond}"
             Height="40" Width="100"/>
  <Button Name="btnChanger" Content="Changer la couleur de fond"
          Click="Button_Click"/>
 </StackPanel>
</Window>
```

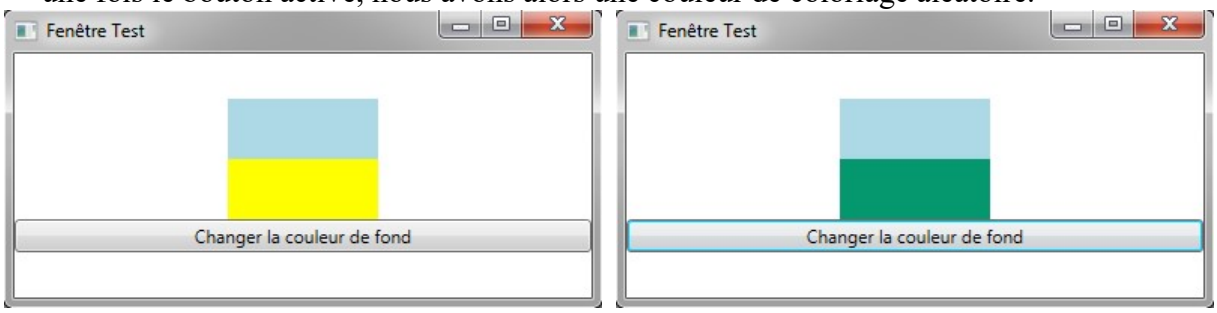
Nous mettons en place une ressource, deux rectangles et un bouton. La ressource permet de définir la couleur de remplissage d'éléments XAML. Les rectangles servent de témoins au comportement de la liaison avec la ressource. Le premier est associé de manière statique (valeur spécifiée au chargement) tandis que le second l'est de manière dynamique (valeur pouvant évoluer en cours d'exécution). Nous avons également créé une méthode `Window_Loaded` dans le code behind pour agir sur la couleur du second bouton au chargement de la fenêtre. Pour terminer, le bouton permet de modifier la valeur associée à la ressource en déterminant aléatoirement une couleur.

Pour ce qui est du code behind, nous avons les instructions suivantes.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
 {Panneau.Resources["RectangleFond"] = Brushes.Yellow;
 }
private void Button_Click(object sender, RoutedEventArgs e)
 {Random rnd = new Random();
  byte r = (byte)rnd.Next(0, 256);
  byte g = (byte)rnd.Next(0, 256);
  byte b = (byte)rnd.Next(0, 256);
  Panneau.Resources["RectangleFond"]
   = new SolidColorBrush(Color.FromRgb(r, g, b));
 }
```

Nous avons les résultats suivants,

- une fois la page chargée, nous pouvons constater que, comme défini dans la méthode de chargement, le second rectangle est coloré en jaune,
- une fois le bouton activé, nous avons alors une couleur de coloriage aléatoire.



Les ressources peuvent être définies en plusieurs endroits. Reprenons-les en illustrant chacune des possibilités.

La racine d'une application peut se voir attribuer des ressources. Il suffit donc d'ajouter dans la balise `<Application >` une section `Resources`. Complétons le code de la section 0.2.4 en spécifiant explicitement une ressource.

```
<Application
  x:Class="SandBox.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
 <Application.Resources>
  <SolidColorBrush x:Key="App_CouleurVert" Color="Green" />
 </Application.Resources>
</Application>
```

Cela signifie que tout objet present dans l'application peut utiliser cette resource. Ainsi, supposons que le code suivant soit intégré dans une page quelconque de l'application, le bouton qui y est défini bénéficie de la ressource déclarée dans la balise `<Application >`.

```
<Button Name="btnVert" Content="Vert"
        Background="{StaticResource App_CouleurVert}"/>
```

Nous pouvons faire de même avec la balise `<Window >` et la portée de la ressource se voit alors naturellement limitée à la page dans laquelle elle est définie.

En fait, tout élément XAML possède la propriété `Resources`. Cela signifie que nous pouvons créer une ressource locale, c'est-à-dire uniquement disponible pour l'élément et sa *descendance*. Même si rien n'est vraiment neuf par rapport aux deux autres cas, si ce n'est la portée, nous pouvons toutefois écrire, pour utiliser dans la ressource définie dans .

```
<StackPanel Name="Panneau" Orientation="Vertical">
 <StackPanel.Resources>
  <SolidColorBrush x:Key="Elt_CouleurVert" Color="Green" />
 </StackPanel.Resources>
 <Button Name="btnVert" Content="Vert"
         Background="{StaticResource Elt_CouleurVert}"/>
</StackPanel>
```

Passons au dernier cas qui consiste à définir une partie des ressources en dehors de l'application WPF. Cette possibilité permet de partager des ressources entre des applications et facilite également la localisation des applications.

La première remarque concerne le `ResourceDictionary`. En effet, ce dernier ne dispose pas d'attribut `x:Key`. Cependant, une référence d'un `ResourceDictionary` dans une collection `MergedDictionaries` est traitée de manière particulière, via la propriété `Source` destinée à enregistrer la localisation *fichier* de la ressource.

Il suffit d'ajouter au projet un nouvel élément de type Dictionnaire de ressources et de compléter le fichier ainsi obtenu que nous baptiserons FichierRessources. Nous pouvons écrire, de manière originale,

```
<ResourceDictionary
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
 <SolidColorBrush x:Key="Ext_CouleurVert" Color="Green" />
</ResourceDictionary>
```

Il nous reste maintenant à utiliser cette ressource extérieure.

```
<Button Name="btnVert" Content="Vert"
        Background="{StaticResource Ext_CouleurVert}">
 <Button.Resources>
  <ResourceDictionary>
   <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="FichierRessources.xaml"/>
   </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
 </Button.Resources>
</Button>
```

Notons que nous pouvons alors utiliser *localement* toutes les ressources définies dans ce fichier externe.

3. Les événements et les commandes

3.1. Introduction

Dans les applications fenêtrées traditionnelles, les évenements associés aux contrôles peuvent être liés à des délégués qui ne sont rien d'autre qu'un *pointeur* vers la méthode qui *gère* l'événement.

Ainsi, comme c'est souvent le cas, nous nous penchons sur l'événement clic associé à un bouton que nous baptisons `MonBouton`.

Dans le fichier d'extension Designer.cs associé à la fenêtre, nous pouvons trouver le code suivant qui est habituellement généré automatiquement lorsque, en mode design, nous déposons un tel bouton sur la fenêtre sur lequel nous double cliquons pour lui associer une méthode.

```
private System.Windows.Forms.Button MonBouton;
MonBouton.Click += new System.EventHandler(this.MonBouton_Click);
```

Le code associé à l'événement se trouve enregistré dans le fichier d'extension cs sous la formesuivante.

```
private void MonBouton_Click(object sender, EventArge e)
{//Code associé au clic sur MonBouton
}
```

3.2. Possibilités supplémentaires en WPF

En wpf, les possibilités de gérer les évenements sont plus nombreuses. Nous pouvons relever 3 modes de *routage*, à savoir

- Direct est identique à ce qui se passe en programmation WinForm standard, à savoir que c'est l'élément qui gère l'événement.
- Bubbling permet, telle une bulle d'air dans l'eau, de remonter dans la hiérarchie des objets XAML où se trouve l'élément à la source de l'événement pour essayer de trouver un gestionnaire d'événement associé.
- Tunneling agit exactement à l'inverse en descendant dans la hiérarchie.

Certains contrôles sont de type Bubbling tandis que d'autres sont de type Tunneling. De plus, ils vont souvent de pair.

Par exemple, `KeyDown`, `MouseDown` et `MouseWheel` (de type Bubbling) sont associés, respectivement, à `PreviewKeyDown`, `PreviewMouseDown` et `PreviewMouseWheel` (de type Tunneling).

Pour se rendre compte du comportement de ces éléments, nous pouvons mettre en place le code XAML suivant.

```
<Window x:Class="SandBox.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Routed Events" Height="300" Width="500">
<Grid x:Name="Grille">
 <Grid.Resources>
  <DataTemplate x:Key="RoutedEventTemplate">
   <TextBlock Text="{Binding Path=RoutedEvent}" Width="auto"
              Margin="0" />
   </DataTemplate>
   <DataTemplate x:Key="SenderTemplate">
    <TextBlock Text="{Binding Path=Sender}" Width="auto" Margin="0"/>
   </DataTemplate>
   <DataTemplate x:Key="ArgsSourceTemplate">
    <TextBlock Text="{Binding Path=ArgsSource}" Width="auto"
               Margin="0"/>
   </DataTemplate>
   <DataTemplate x:Key="SourceTemplate">
    <TextBlock Text="{Binding Path=Source}" Width="auto" Margin="0"/>
   </DataTemplate>
  </Grid.Resources>
  <Grid.RowDefinitions>
   <RowDefinition Height="auto"/>
   <RowDefinition Height="auto"/>
   <RowDefinition Height="100*"/>
  </Grid.RowDefinitions>
  <StackPanel Orientation="Horizontal" HorizontalAlignment="Left">
   <Button x:Name="btnBouton" Margin="10" Padding="2" Content="Bouton"
           Height="auto"/>
   <Button x:Name="btnRAZ" Margin="10" Padding="2" Content="R à Z"
           Height="auto" Click="btnRAZ_Click"/>
  </StackPanel>
  <ListView x:Name="lvResultats" Margin="0,0,0,0"
            IsSynchronizedWithCurrentItem="True" Grid.Row="2" >
   <ListView.View>
    <GridView>
     <GridViewColumn CellTemplate="{StaticResource RoutedEventTemplate}"
                     Header="RoutedEvent" Width="150"/>
     <GridViewColumn CellTemplate="{StaticResource SenderTemplate}"
                     Header="Sender" Width="100"/>
     <GridViewColumn CellTemplate="{StaticResource ArgsSourceTemplate}"
                     Header="ArgsSource" Width="100"/>
     <GridViewColumn CellTemplate="{StaticResource SourceTemplate}"
                     Header="Source" Width="100"/>
    </GridView>
   </ListView.View>
  </ListView>
 </Grid>
</Window>
```

L'élément de type `ListView` voit ses colonnes définies par l'intermédiaire des balises `GridViewColumn` pour lesquelles le remplisssage est pris en charge par des ressources définies au niveau de `Grille`. Ces ressources sont de type `DataTemplate` et sont composées d'un `TextBlock` qui récupère les données par l'intermédiaire du code C# qui suit et des propriétés adéquates qui y sont définies.

La classe sur laquelle s'appuie la liaison est simplement une *énumération* des propriétés, à savoir.

```
public class EventDemoClass
 {
  public string RoutedEvent { get; set; }
  public string Sender { get; set; }
  public string ArgsSource { get; set; }
  public string Source { get; set; }
 }
```

Pour ce qui est du code c# associé à la fenpêtre, après avoir associé les éléments XAML `Grille`, `btnBouton` et `lvResultats` à une méthode générique de traitement des événements, nous définissons la méthode en elle-même qui ne fait que remplir `lvResultats` selon l'événement déclenché en utilisant la méthode `OterNamespace` qui permet de récupérer le type seul et nous terminons par la méthode qui permet de vider `lvResultats`.

```
public partial class MainWindow : Window
{public MainWindow()
 {InitializeComponent();
  UIElement[] els = { this, Grille, btnBouton, lvResultats };
  foreach (UIElement el in els)
  {//keyboard
   el.PreviewKeyDown += TraitementGenerique;
   el.PreviewKeyUp += TraitementGenerique;
   el.PreviewTextInput += TraitementGenerique;
   el.KeyDown += TraitementGenerique;
   el.KeyUp += TraitementGenerique;
   el.TextInput += TraitementGenerique;
   //Mouse
   el.MouseDown += TraitementGenerique;
   el.MouseUp += TraitementGenerique;
   el.PreviewMouseDown += TraitementGenerique;
   el.PreviewMouseUp += TraitementGenerique;
   el.AddHandler(Button.ClickEvent,
                 new RoutedEventHandler(TraitementGenerique));
  }
 }
 private void TraitementGenerique(object sender, RoutedEventArgs e)
 {lvResultats.Items.Add(new EventDemoClass()
                    {RoutedEvent = e.RoutedEvent.Name,
                     Sender = OterNamespace(sender),
                     ArgsSource = OterNamespace(e.Source),
                     Source = OterNamespace(e.OriginalSource)});
 }
 private string OterNamespace(object obj)
 {string[] astr = obj.GetType().ToString().Split('.');
  return astr[astr.Length - 1];
 }
 private void btnRAZ_Click(object sender, RoutedEventArgs e)
 {lvResultats.Items.Clear();}
}
```

Une fois le code mis en place, nous pouvons constater la hiérarchie suivante

- `Window`,
  - `Grid`,
    - `StackPanel` contenant les deux objets de type `Button`,
  - `ListView`.

Si nous cliquons sur la fenêtre, en dehors de tout élément, nous constatons l'affichage des événements `PreviewMouseDown`, `MouseDown`, `PreviewMouseUp` et `MouseUp` uniquement relatifs à `Window`.

Si nous cliquons sur le premier bouton (`btnBouton`), nous pouvons voir que les événements `PreviewMouseDown` et `PreviewMouseUp` parcourent la hiérarchie depuis la racine (`Window`, `Grid`, `Button`) tandis que l'événement `Click` fait le parcours inverse (`Button`, `Grid`, `Window`).

3.3. Création d'événements

Nous allons illustrer la création d'événements personnalisés par l'intermédiaire d'un contrôle qui l'est tout autant.

Pour ce faire, nous pouvons ajouter un nouvel élément à notre projet via un clic droit dans l'explorateur de solutions puis spécifier l'élément comme étant un Contrôle Utilisateur.

Nous y plaçons un label et deux boutons dans un StackPanel de la manière suivante.

```
<UserControl x:Class="SandBox.CtrlEvenement"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  mc:Ignorable="d">
 <Grid>
  <Grid.RowDefinitions>
   <RowDefinition Height="Auto"/>
   <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <Label Content="Deux boutons" HorizontalAlignment="Center"/>
  <StackPanel Grid.Row="2" Orientation="Horizontal">
   <Button x:Name="btnBouton" Margin="10,0,10,0"
           Content="Clic standard"/>
   <Button x:Name="btnBoutonArgument" Margin="10,0,10,0"
           Content="Clic Arguments"/>
  </StackPanel>
 </Grid>
</UserControl>
```

Pour ce qui est du code behind, nous commencçons par créer une classe destinée à prendre en charge l'événement avec paramètre. Cette classe hérite de la classe `RoutedEventArgs` et possède un argument de type entier associé au paramètre.

Nous pouvons écrire

```
public class EvenementArgument : RoutedEventArgs
{public int Nombre { get; private set; }
 public EvenementArgument(RoutedEvent routedEvent, int nombre)
  : base(routedEvent)
 {Nombre = nombre;}
}
```

Une fois cette classe en place, nous pouvons penser à enregistrer l'événement, de type `Bubble`, à l'aide de la classe `EventManager`. Signalons que cette méthode possède plusieurs surcharges.

```
public static readonly RoutedEvent EvenementClic =
                                    EventManager.RegisterRoutedEvent(
                                    "Clic", RoutingStrategy.Bubble,
                                    typeof(RoutedEventHandler),
                                    typeof(CtrlEvenement));
```

En passant, nous écrivons alors le code permettant aux contrôles de souscrire à l'événement.

```
public event RoutedEventHandler CustomClick
{add { AddHandler(EvenementClic, value); }
 remove { RemoveHandler(EvenementClic, value); }
}
```

Nous pouvons également mettre en place la méthode associée au clic sur `btnBouton`. Cette méthode redirigera vers l'événement souhaité, à savoir `EvenementClic`.

```
private void btnBouton_Click(object sender, RoutedEventArgs e)
{RaiseEvent(new RoutedEventArgs(EvenementClic));}
```

En ce qui concerne l'événement avec paramètres, la façon de faire est similaire. Il suffit de prévoir, au sein du contrôle utilisateur, une variable pour *dialoguer* avec le champ enregistré dans la classe `EvenementArgument` et de placer un pointeur de fonction (`delegate`) dans le code du contrôle personnalisé.

```
public delegate void EvenementClicPointeur(object sender,
                                           EvenementArgument e);
public int ClicCompteur { get; private set; }
```

Ensuite, nous adaptons le code precedent sous la forme suivante.

```
public static readonly RoutedEvent EvenementClicArgument =
                        EventManager.RegisterRoutedEvent(
                        "ClicArgument", RoutingStrategy.Bubble,
                        typeof(EvenementClicPointeur),
                        typeof(CtrlEvenement));
public event EvenementClicPointeur CustomClickWithCustomArgs
{add { AddHandler(EvenementClicArgument, value); }
 remove { RemoveHandler(EvenementClicArgument, value); }
}
private void btnBoutonArgument_Click(object sender, RoutedEventArgs e)
{
 RaiseEvent(new EvenementArgument(EvenementClicArgument,
            ++ClicCompteur));
}
private void btnBouton_Click(object sender, RoutedEventArgs e)
{ RaiseEvent(new RoutedEventArgs(EvenementClic)); }
```

Il ne nous reste plus qu'à mettre tout ce code en function. Après avoir déposé le contrôle utilisateur sur la fenêtre, nous pouvons compléter le constructeur de la manière suivante.

```
public CtrlEvenement()
{InitializeComponent();
 btnBouton.Click += new RoutedEventHandler(btnBouton_Click);
 btnBoutonArgument.Click +=
  new RoutedEventHandler(btnBoutonArgument_Click);
}
```

3.4. Les `RoutedCommand`

Le système de commande en WPF est base sur les objets de types `RoutedEvent` et `RoutedCommand`.

Outer le fait que le système de commande soit repris dans d'autres environnements de développements, il intègre une différence fondamentale par rapport à la gestion traditionnelle des événements en ce sens qu'il sépare la logique de la sémantique et du déclencheur.

Ainsi, plusieurs sources peuvent invoquer la même logique d'instructions tout en étant que cette logique peut également être adaptée à chaque source.

Un exemple de cette manière de faire : les commandes Copier, Couper, Coller qui se retrouvent dans de nombreux contextes différents mais réalisent toujours la même *action*.

Il existe plusieurs types de commandes, à savoir

- `ApplicationCommands` qui reprennent, entre autres, `Copy`, `Cut` et `Paste`,
- `MediaCommands` qui contiennent, entre autres, `ChannelUp`, `ChannelDown` et `VolumeMute`,
- `NavigationCommands` qui voient `BrowseBack` et `BrowseForward` parmi les commandes disponibles,
- `ComponentCommands` qui contiennent, par exemple, `MoveDown`, `MoveFocusPage` et `MoveToEnd`,
- `EditingCommands` qui reprennent `AlignCenter`, `Backspace` et `Delete`.

Avec ces commandes fournies, il est possible de créer des fonctionnalités élégantes avec un minimum de code. Prenons le cas le plus simple qui consiste à utiliser les commandes de type `EditingCommands` en vue de créer un éditeur de texte.

Signalons que pour cette application, tout se passe dans le code XAML et aucune ligne C# ne doit être écrite.

Nous avons

```
<Window x:Class="SandBox.Editeur"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Editeur" Height="300" Width="300" ResizeMode="NoResize">
 <StackPanel Orientation="Vertical" Width="auto">
  <StackPanel Orientation="Horizontal" Margin="10" Height="40">
   <Button Command="ApplicationCommands.Cut" Content ="Couper"
           Margin="5" CommandTarget="{Binding ElementName=tbEditeur}" />
   <Button Command="ApplicationCommands.Copy" Content="Copier"
           Margin="5" CommandTarget="{Binding ElementName=tbEditeur}" />
   <Button Command="ApplicationCommands.Paste" Content="Coller"
           Margin="5" CommandTarget="{Binding ElementName=tbEditeur}" />
   <Button Command="ApplicationCommands.Undo" Content="Annuler"
           Margin="5" CommandTarget="{Binding ElementName=tbEditeur}" />
   <Button Command="ApplicationCommands.Redo" Content="Rétablir"
           Margin="5" CommandTarget="{Binding ElementName=tbEditeur}" />
  </StackPanel>
  <TextBox x:Name="tbEditeur" HorizontalAlignment="Stretch" Margin="10"
           MaxLines="60" Height="190" TextWrapping="Wrap"
           Background="#FFF1FFB2"/>
 </StackPanel>
</Window>
```

Voyons maintenant comment définir des `RoutedCommand` personnalisée. Le principe est simple et naturel :

- Déclarer un objet de type `RoutedCommand`,
- Créer un objet de type `CommandBinding` pour permettre une référence à l'objet de type `RoutedCommand`,
- Associer ce qui a été créé à des contrôles XAML.

Dans la classe `ClasseCommande`, nous allons définir un objet de type `RoutedCommand` pour lequel l'accès doit être permis via l'espace de nom `System.Windows.Input`.

```
public class ClasseCommande
{public static readonly RoutedCommand Commande
  = new RoutedCommand("Commande", typeof(ClasseCommande));
}
```

Nous allons ensuite créer un contrôle utilisateur qui ne contient qu'un seul bouton. Nous faisons référence à l'espace de nom `SandBox` pour renseigner la commande comme étant `Commande` que nous venons de définir comme membre de `ClasseCommande`.

```
<UserControl x:Class="SandBox.CtrlCommande"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:SandBox">
 <Button Margin="5" Command="{x:Static local:ClasseCommande.Commande}"
         Content="Cliquer si possible"/>
</UserControl>
```

Ensuite, nous pouvons créer une nouvelle fenêtre. Nous y plaçons évidemment le contrôle utilisateur `CtrlCommande` ainsi qu'un élément XAML de type `TexBox`. Nous choisissons de mettre ces deux contrôles dans un `StackPanel` que nous centrons sur la fenêtre.

Dans l'en-tête de cette même fenêtre, nous ajoutons la balise `CommandBinding` pour préparer la liaison avec la commade définie auparavant et en n'oubliant pas de faire référence à l'espace de nom associé. Nous avons le code XAML suivant.

```
<Window x:Class="SandBox.WindowCommande"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="WindowCommande" Height="300" Width="300"
  xmlns:local="clr-namespace:SandBox">
 <Window.CommandBindings>
  <CommandBinding Command="{x:Static local:ClasseCommande.Commande}"
                  CanExecute="Commande_CanExecute"
                  Executed="Commande_Executed" />
 </Window.CommandBindings>
 <StackPanel Orientation="Vertical"
             HorizontalAlignment="Center" VerticalAlignment="Center">
  <TextBox Name="tbARemplir" Width="100" Height="30" />
  <local:CtrlCommande x:Name="MonControle" />
 </StackPanel>
</Window>
```

Pour terminer, il nous suffit de passer au code C# en intégrant deux nouvelles méthodes.

```
private void Commande_CanExecute(object sender,
                                 CanExecuteRoutedEventArgs e)
{e.CanExecute = !(string.IsNullOrEmpty(tbARemplir.Text));}
private void Commande_Executed(object sender, ExecutedRoutedEventArgs e)
{MessageBox.Show(tbARemplir.Text);}
```

4. Les propriétés

Les propriétés (CLR) telles que nous les connaissons en programmation orientée objet rendent habituellement accessibles des données membres de type `private` en permettant la mise en place de vérifications, essentiellement à l'écriture.

Le concept abordé dans ce chapitre, à savoir les propriétés de dépendance, permet d'étendre ces possibilités. L'introduction de WPF a été l'occasion de mettre en place un système de propriétés permettant

- de notifier les modifications,
- valider les valeurs transmises,
- hériter des valeurs transmises,
- participer à la liaison de données, aux animations, aux styles et aux templates.

4.1. Les propriétés de dépendance

La mise en place de telles propriétés est similaire à celle utilisée pour les `RoutedEvent` abordés précédemment, à savoir

- déclarer un objet de type `DepedencyProperty`,
- l'initialiser,
- mettre en place les propriétés *classiques* dans lesquelles il est déconseillé d'ajouter du code propre (se limiter à `GetValue`/`SetValue`).

Nous créons une nouvelle classe, par exemple, `MaClasse` que nous devons faire hériter d'un contrôle XAML, `Window` par exemple. Nous y intégrons une donnée membre de type `DependencyProperty`, par exemple `propriete` de la manière suivante.

```
class ClasseDependance : Window
{public static readonly DependencyProperty propriete
  = DependencyProperty.Register("ProprieteDependance",
                                typeof(float),
                                typeof(ClasseDependance));
}
```

Par l'intermédiaire de cette instruction, nous signalons que tout objet de type `ClasseDependance` disposera d'une `DependencyProperty` de type `float` dont le nom sera `ProprieteDependance` et pourra être utilisé tel quel dans le code XAML. Pour terminer, `propriete` est une clé qui permet de référencer cette `DependencyProperty` depuis le code pour permettre d'effectuer une opération la concernant. Signalons toutefois qu'il est également possible d'y accéder par l'intermédiaire de son nom `ProprieteDependance`.

Pour rendre utilisable cette propriété de manière totalement invisible, nous pouvons définir la propriété (CLR) suivante.

```
public float ProprieteDependance
{get{return (float)GetValue(propriete);}
 set{SetValue(propriete,value);}
}
```

Signalons au passage que nous sommes passés à un typage de type fort puisque, en utilisant une `DependencyProperty`, ce typage ne se fait qu'à l'éxécution.

4.2. Les propriétés attachées

Il faut savoir qu'une propriété de dépendance d'une instance n'occupe pas de mémoire tant que cette propriété n'a pas été modifiée renvoyant, dans ce cas, la valeur par défaut qu'il est d'ailleurs possible de préciser lors de l'appel à `DependencyProperty.Register`.

Une propriété attachée est une `DependencyProperty` attachée à n'importe quel objet XAML. Prenons l'exemple de la propriété `Dock`. Cette propriété, commune à tous les contrôles, ne sera utilisée que lorsque le contrôle est placé dans un élément de type `DockPanel`. Comme rares sont les contrôles qui seront dans cette situation, la propriété est rendue accessible à ces derniers mais n'occupe pas de place superflue pour les autres.

De plus, il devient possible d'ajouter des propriétés à la classe `Control` dont nous ne sommes pas l'auteur.

Pour déclarer une telle propriété, le mécanisme est similaire à celui découvert au point précédent si ce n'est que le processus des propriétés CLR n'est pas disponible dans ce contexte. Nous écrivons

```
public class ClasseAttachee
{public static readonly DependencyProperty propriete
  = DependencyProperty.RegisterAttached("ProprieteAttachee",
                                        typeof(double),
                                        typeof(ClasseAttachee));
 public static double GetProprieteAttachee(DependencyObject target)
 {return (double)target.GetValue(propriete);}
 public static void SetProprieteAttachee(DependencyObject target,
                                         double value)
 {target.SetValue(propriete, value);}
}
```

Adaptons ce code à un cas concret où nous souhaitons placer une bannière au dessus de toute fenêtre selon une valeur logique qui témoigne de l'affichage ou non. La bannière est une simple image (LARA.bmp) que nous pouvons importer dans le projet.

Nous déclarons donc la classe `ProprieteAttachee` dand laquelle nous plaçons un objet `IsBanniere` de type `DependencyProperty`. Ensuite, nous définissons les méthodes de lecture `GetIsBanniere` et d'écriture `SetIsBanniere` associé à cet objet. Nous avons

```
public class ProprieteAttachee
{public static readonly DependencyProperty IsBanniere
  = DependencyProperty.RegisterAttached("IsBanniereNom",
                    typeof(Boolean),
                    typeof(ProprieteAttachee),
                    new FrameworkPropertyMetadata(IsBanniereModifier));
 public static void SetIsBanniere(DependencyObject obj, Boolean value)
 {obj.SetValue(IsBanniere, value);}
 public static Boolean GetIsBanniere(DependencyObject obj)
 {return (Boolean)obj.GetValue(IsBanniere);}
}
```

La seule différence avec ce qui précède se trouve dans l'instanciation de `IsBanniere`.

Ensuite, nous définissons la méthode qui permet d'afficher l'image en haut de la fenêtre. Pour ce faire, nous mettons en place un événement `RoutedEvent` de type *fenêtre chargée* que nous déclencherons par ailleurs. Le principe de cette méthode est d'utiliser un `DockPanel` de manière à ce que les composants définis effectivement dans la fenêtre puissent prendre toute la place laissée par l'image, en dessous de cette dernière.

```
public static void wnd_Loaded(object sender, RoutedEventArgs e)
{try
 {DockPanel dp = new DockPanel();
  dp.LastChildFill = true;
  StackPanel sp = new StackPanel();
  dp.Children.Add(sp);
  sp.Background = new SolidColorBrush(Colors.CornflowerBlue);
  sp.Orientation = Orientation.Vertical;
  sp.SetValue(DockPanel.DockProperty, Dock.Top);
  BitmapImage bitmap = new BitmapImage(new Uri("LARA.bmp",
                                       UriKind.Relative));
  Image image = new Image();
  image.Source = bitmap;
  sp.Children.Add(image);
  UIElement el = ((DependencyObject)sender as Window).Content
                 as UIElement;
  el.SetValue(DockPanel.DockProperty, Dock.Bottom);
  ((DependencyObject)sender as Window).Content = null;
  dp.Children.Add(el);
  ((DependencyObject)sender as Window).Content = dp;
 }
 catch (Exception ex)
 {System.Diagnostics.Debug.WriteLine("Exception : " + ex.Message));}
}
```

Pour en terminer avec la classe `ProprieteAttachee`, il nous suffit de prévoir l'association de la fenêtre appelante avec l'événement de chargement que nous venons de définir. La méthode suivante permet effectivement de reconnaître l'éventuelle fenêtre appelante et lui associe alors l'événement `wnd_Loaded`.

```
public static void IsBanniereModifier(DependencyObject obj,
                    DependencyPropertyChangedEventArgs args)
{if ((bool)args.NewValue)
  if (obj is Window)
  {Window fen = (Window)obj;
   fen.Loaded += new RoutedEventHandler(wnd_Loaded);
  }
}
```

Il ne reste plus qu'à récupérer les fruits de notre travail. Pour ce faire, dans n'importe quelle fenêtre, nous ajoutons les lignes suivantes dans l'en-tête.

```
xmlns:local="clr-namespace:SandBox"
local:ProprieteAttachee.IsBanniere="true"
```

La valeur `true`, transmise par l'intermédiaire de `SetIsBanniere` et, par conséquent, de `IsBanniereModifier` entraîne l'association à la méthode `wnd_Loaded` et implique donc l'affichage de la bannière qui n'apparaîtra pas si la valeur utilisée est `false`.

Nous pourrions également remplacer cette bannière par un élément contenant le menu commun à toutes les fenêtres de l'application sur lequel nous pourrions interagir avec des événements associés de type `RoutedCommand`.

5. Databinding

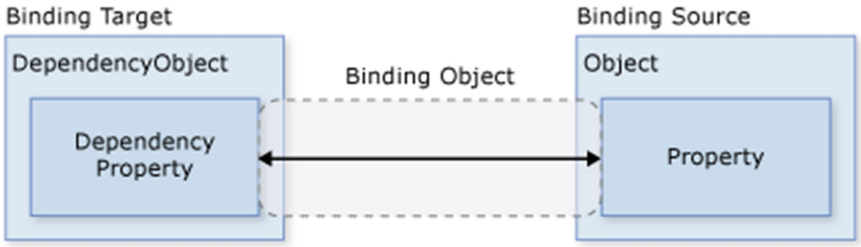
5.1. Introduction

Ce concept n'est pas neuf en C#. Nous le retrouvons dans les applications fenêtres traditionnelles (WinForm) ainsi que dans les sites web (ASP.Net).

Le principe de base est simple. D'un côté, il y a les données et, de l'autre, il y a l'écran. La liaison de données est destinée à mettre ces deux entités en contact.

Cependant, WPF va plus loin dans la liaison des données avec l'interface utilisateur en mettant à la disposition des développeurs le schéma de base suivant :

Les outils mis à disposition du développeur ont évolués pour rendre cette idée de base plus performante.



Chaque liaison repose sur

- un objet cible de la liaison,
- une propriété cible,
- une source de liaison,
- un chemin d'accès vers la valeur de la source de liaison.

La source de données peut évidemment être définie dans le code behind mais également dans l'interface utilisateur.

La propriété cible doit être une propriété de dépendance telle que nous l'avons définie précédemment.

Appuyons-nous sur un exemple de type Hello World *paramétré*. Le but est de faire apparaître dans un `TextBlock` le contenu de ce que l'utilisateur tape dans une zone d'encodage de type `TextBox`.

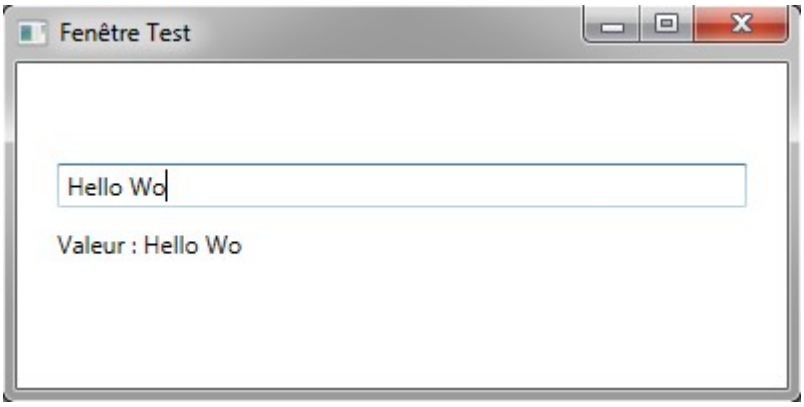
Pour ce faire, nous écrivons par exemple

```
<StackPanel Margin="20,50,20,50">
 <TextBox Name="Source" />
 <WrapPanel Margin="0,10">
  <TextBox Name="Source" Text="Valeur : " />
  <TextBlock Name="Cible"
             Text="{Binding Path=Text, ElementName=Source}" />
 </WrapPanel>
</StackPanel>
```

Nous retrouvons dans le code suivant les éléments suivants.

- un objet cible de la liaison : `Cible`,
- une propriété cible : `Text`,
- une source de liaison : `Source`,
- un chemin d'accès vers la valeur de la source de liaison : `Text`.

Nous obtenons le remplissage du composant `Cible`, de type `TextBlock`, au fur et à mesure que le composant `Source`, de type `TextBox`, est modifié.



5.2. Les convertisseurs

Les données manipulées nécessitent parfois une conversion pour être affichée selon un format adéquat. XAML met à la disposition des développeurs un processus de conversion.

Ainsi, dans l'exemple suivant, nous allons mettre en place un composant de type `Slider` qui va, par liaison, permettre de faire tourner un composant de type `Rectangle` en prenant des valeurs comprises entre 0° et 360°. Le code XAML peut prendre la forme suivante où nous définissons une ressource locale à la grille définie.

```
<Grid>
 <Grid.RowDefinitions>
  <RowDefinition Height="*" />
  <RowDefinition Height="Auto" />
  <RowDefinition Height="Auto" />
 </Grid.RowDefinitions>
 <Grid.Resources>
  <RotateTransform x:Key="rota" Angle="0" />
 </Grid.Resources>
 <Rectangle Fill="Red" Width="100" Height="50"
            LayoutTransform="{StaticResource rota}" />
 <Slider Grid.Row="2" Minimum="0" Maximum="360"
         Value="{Binding Source={StaticResource rota}, Path=Angle}" />
</Grid>
```

Nous terminons cette manipulation en plaçant un composant de type `TextBox` pour afficher, en radians, la valeur déterminée par le `Slider`. Comme nous utilisons un `TextBox`, nous souhaitons permettre à l'utilisateur d'encoder une valeur au clavier et d'en répercuter les effets au niveau du `Slider` et, par conséquent, du `Rectangle`. La conversion se fera donc dans les deux sens.

Au niveau du code XAML, nous renseignons, dans l'en-tête de la page, l'espace de noms où se trouve la classe `Degre2RadianConversion` via l'instruction suivante.

```
xmlns:local="clr-namespace:SandBox"
```

Nous ajoutons alors, comme ressource locale à la grille, l'instruction suivante qui fait référence à la fonction `Degre2RadianConversion` définie par après dans le code behind.

```
<local:Degre2RadianConversion x:Key="conv" />
```

Ensuite, nous plaçons le composant `TextBox` associé à la ressource `rota` par l'intermédiaire du convertisseur `conv` que nous venons de définir en ressource.

```
<TextBox Grid.Row="1" TextAlignment="Center"
         Text="{Binding Source={StaticResource rota}, Path=Angle,
                        Converter={StaticResource conv}}" />
```

Pour terminer, il nous reste à définir cette conversion dans le code behind. D'une part, la conversion degrés vers radians se fait simplement en multipliant la valeur reçue du `Slider`, castée d'objet en double, par $\frac{\pi}{180}$. Pour la conversion inverse, le principe est identique. Toutefois, la valeur encodée doit être comprise entre 0 et 2π. Nous avons le code suivant.

```
public class Degre2RadianConversion : IValueConverter
 {public object Convert(object valeur, Type typecible,
                        object parametre, CultureInfo culture)
   {return (double)valeur * Math.PI / 180;
   }
  public object ConvertBack(object valeur, Type typecible,
                            object parametre, CultureInfo culture)
   {double Angle = double.Parse(valeur.ToString());
    if (Angle < 0) return 0;
    else if (Angle > 2 * Math.PI) return 360;
    else return Angle * 180 / Math.PI;
   }
 }
```

En WPF, plusieurs voies peuvent mener au même résultat. Ainsi, si nous baptisons `barre` le composant de type `Slider`.

```
<Rectangle Fill="Yellow" Width="50" Height="100">
 <Rectangle.RenderTransform>
  <RotateTransform Angle="{Binding ElementName=barre, Path=Value}"
                   CenterX="25" CenterY="50" />
 </Rectangle.RenderTransform>
</Rectangle>
```

L'accès au convertisseur peut se faire comme suit

```
<TextBox Grid.Row="3" TextAlignment="Center"
         Text="{Binding ElementName=barre, Path=Value,
                Converter={StaticResource conv}}" />
```

5.3. La propriété `DataContext`

Nous abordons ici une des propriétés fondamentales dont disposent la majorité des éléments XAML. Cette propriété permet aux éléments d'hériter, de leurs parents, des informations relatives à la source de données comme le chemin d'accès. Cet héritage est évidemment occulté dans le cas où l'élement redéfinit son propre contexte.

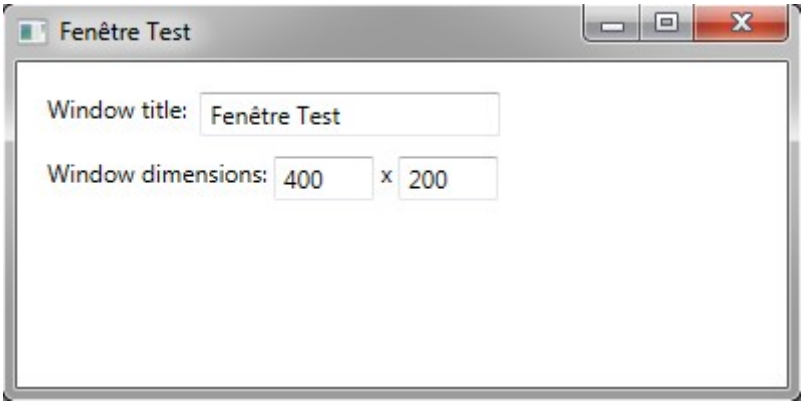
Cette propriété peut être associée directement avec un objet C#. L'instruction ci-dessous, intégrée dans le constructeur de la fenêtre, définit la fenêtre elle-même comme source de données.

```
this.DataContext = this;
```

Le code XAML peut alors être inséré dans une fenêtre.

```
<StackPanel Margin="15">
 <WrapPanel>
  <TextBlock Text="Window title:  " />
  <TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}"
           Width="150" />
 </WrapPanel>
 <WrapPanel Margin="0,10,0,0">
  <TextBlock Text="Window dimensions: " />
  <TextBox Text="{Binding Width}" Width="50" />
  <TextBlock Text=" x " />
  <TextBox Text="{Binding Height}" Width="50" />
 </WrapPanel>
</StackPanel>
```

Le résultat est le suivant où la liaison peut s'observer en modifiant la taille de la fenêtre, ce qui entraine une actualisation des dimensions affichées..



Signalons, au passage, la présence d'un trigger sur la mise à jour qui permet d'actualiser directement l'affichage du titre de la fenêtre sans devoir attendre que l'on quitte le contrôle `TextBox` contrairement à la largeur et à la longueur lorsque l'utilisateur modifie la taille de la fenêtre. Notons toutefois que cette mise à jour se fait dans les deux sens :

- Le dimensionnement par la souris est répercuté dans les zones de texte,
- La modification des valeurs dans les zones de textes entraîne un redimensionnement de la fenêtre.

Nous revenons en détail sur ces particularités dans le point suivant en présentant les autres possibilisés disponibles et en implémentant dans le code behind l'événement `PropertyChanged`.

Avant de passer à une liaison avec des donnés présentes en code C#, nous allons voir les trois moyens de remplir un composant avec des données se trouvant également sur la page.

Nous commençons par placer notre *source* de données, à savoir un composant de type `TreeView`.

```
<TreeView Name="tvArborescence" Grid.Row="3"
          SelectedItemChanged=" tvArborescence SelectedItemChanged">
 <TreeViewItem Header="Desktop">
  <TreeViewItem Header="Computer">
   <TreeViewItem Header="Personnal PC"/>
   <TreeViewItem Header="Hard Disks"/>
   <TreeViewItem Header="Removable Disks"/>
   <TreeViewItem Header="Printers, Scanners, ..."/>
  </TreeViewItem>
  <TreeViewItem Header="Recycle Bin" />
  <TreeViewItem Header="Control Panel">
   <TreeViewItem Header="Programs"/>
   <TreeViewItem Header="Security"/>
   <TreeViewItem Header="User Accounts"/>
  </TreeViewItem>
  <TreeViewItem Header="Network">
   <TreeViewItem Header="House"/>
   <TreeViewItem Header="City"/>
   <TreeViewItem Header="World"/>
  </TreeViewItem>
 </TreeViewItem>
</TreeView>
```

Nous plaçons ensuite un composant de type `TextBlock` pour chacun des trois modes de communication présentés. Le premier fait référence à un événement associé à l'élément de type `TreeView`, le deuxième utilise les propriétés XAML et le troisième et dernier fait appel à la liaison de données par l'intermédiaire du code C#.

```
<TextBlock x:Name="tbEvenement" TextAlignment="Center" />
<TextBlock x:Name="tbXAML" TextAlignment="Center"
           Text="{Binding ElementName=treeView, Path=SelectedItem.Header}" />
<TextBlock x:Name="tbBehind" TextAlignment="Center" />
```

Le code C# ne concerne évidemment que les premier et troisième composants.

En ce qui concerne `tbEvenement`, nous avons choisi, comme indiqué dans la définition de `tvArborescence`, d'utiliser l'évément `SelectedItemChanged` qui se produit à chaque nouvelle sélection de l'utilisateur.

```
private void tvArborescence_SelectedItemChanged(object sender,
              RoutedPropertyChangedEventArgs<object> e)
{
 tbEvenement.Text
  = (tvArborescence.SelectedItem as TreeViewItem).Header.ToString();
}
```

Pour `tbBehind`, nous nous limitons à initialiser la liaison dans le constructeur de la fenêtre.

```
Binding binding = new Binding();
binding.Source = tvArborescence;
binding.Path = new PropertyPath("SelectedItem.Header");
tbBehind.SetBinding(TextBlock.TextProperty, binding);
```

Approfondissons maintenant la liaison en code behind en nous penchant sur des données définies en C#.

5.4. Liaison avec les données du code behind

Nous allons maintenant considérer une liaison avec des données définies dans le code C#.

5.4.1. Liaison sur un objet

Pour ce faire, nous commençons par définir les données à manipuler via une classe `Personne` dans laquelle nous plaçons un identifiant, un nom et un prénom et les constructeurs et accesseurs associés. Nous avons

```
public class Personne
 {public int ID { get; set; }
  public string Nom { get; set; }
  public string Prenom { get; set; }
  public Personne()
   { }
  public Personne(string Nom_, string Prenom_)
   {Nom = Nom_;
    Prenom = Prenom_;
   }
  public Personne(int ID_, string Nom_, string Prenom_)
   :this(Nom_, Prenom_)
   { ID = ID_; }
   }
 }
```

Nous devons maintenant nous pencher sur le code XAML de la fenêtre. Nous commençons par renseigner, dans la définition de la fenêtre, l'exspace de noms défini dans le code behind.

```
xmlns:local="clr-namespace:SandBox"
```

Une fois cet espace de noms rendu accessible depuis le code XAML, nous pouvons accéder aux ressources qui lui sont associées. C'est ainsi que nous définissons un `DataContext` propre à la fenêtre par le code XAML suivant.

```
<Window.DataContext>
 <local:Personne ID="2" Nom="Ovronnaz" Prenom="Simon" />
</Window.DataContext>
```

Nous renseignons la classe que nous utilisons et instancions un objet de manière à visualiser, par la suite, les données dès la conception de l'interface utilisateur.

Nous pouvons maintenant passer aux contrôles, de type `TextBlock`, associés à ces données. Nous pouvons écrire

```
<StackPanel Name="Bibi" Margin="10">
 <TextBlock Text="{Binding Path=ID}" />
 <TextBlock Text="{Binding Path=Prenom}" />
 <TextBlock Text="{Binding Path=Nom}" />
</StackPanel>
```

Signalons que par l'utilisation de ces instructions XAML, nous pouvons remplir nos contrôles et visualiser, dès la conception, l'effet de ce remplissage.

Nous pouvons compléter le code behind en déclarant et en instanciant un objet de type `Personne` et en spécifiant le `DataContext` de l'objet `Bibi` précédemment déclaré.

Ainsi, nous avons déclaré le quatuor annoncé en 5.1, à savoir l'objet cible de type `TextBlock`, la propriété cible de type `Text`, la valeur à utiliser `Nom` et l'objet source de type `Personne`.

Nous avons, d'une part,

```
public Personne p;
```

et, d'autre part,

```
public MainWindow()
 {InitializeComponent();
  p = new Personne(1, "Winch", "Largo");
  Bibi.DataContext = p;
 }
```

Une fois l'application exécutée, nous pouvons constater que les données relatives à Simon Ovronnaz sont remplacées par celles de Largo Winch.

Nous allons maintenant lier effectivement l'interface utilisateur et les données qui sont définies dans le code behind pour que les modifications se fassent dans les deux sens, à savoir à partir de l'interface utilisateur ou à partir du code behind.

Pour faire en sorte que toute modification des données soient prises en compte, nous devons faire hériter nos classes de données de l'interface `INotifyPropertyChanged` (dont l'espace de noms est `System.ComponentModel`). Vu cette restriction imposée à toutes nos classes de données, nous créons une classe `Data` qui hérite de cette interface et dans laquelle nous plaçons les ressources nécessaires.

```
public class Data : INotifyPropertyChanged
 {public event PropertyChangedEventHandler PropertyChanged;
  protected virtual void OnPropertyChanged(string propertyName)
   {PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
     handler(this, new PropertyChangedEventArgs(propertyName));
   }
 }
```

Nos classes de données, comme `Personne`, hériteront alors de cette classe `Data`. Les modifications à mettre en place concernent les accesseurs en écriture dans lesquels nous appelons la méthode `OnPropertyChanged`.

Nous écrivons.

```
public int ID
 {
  get { return _ID; }
  set { _ID = value; OnPropertyChanged("ID"); }
 }
public string Nom
 {
  get { return _Nom; }
  set { _Nom = value; OnPropertyChanged("Nom"); }
 }
public string Prenom
 {
  get { return  Prenom; }
  set { _Prenom = value; OnPropertyChanged("Prenom"); }
 }
```

Telle est l'implémentation classique. Toutefois, cette manière de faire est sensible au nom donné à l'accesseur et il ne faut pas oublier de répercuter toute modification à l'argument transmis à la méthode `OnPropertyChanged`.

Nous allons, par le code qui suit, prendre en charge la modification de la valeur du champ traité et la répercussion de ce changement vers l'interface utilisateur. Pour ce faire, nous commençons par agir au niveau de la classe `Data` déjà créée et intégrons le code précédemment écrit à une nouvelle méthode qui prend en charge l'enregistrement de la nouvelle valeur. Nous pouvons écrire le code suivant.

```
public class Data : INotifyPropertyChanged
 {public event PropertyChangedEventHandler PropertyChanged;
  protected bool AssignerChamp<T>(ref T field,
                                  T value,
                                  string propertyName
   {if (EqualityComparer<T>.Default.Equals(field, value)) return false;
    PropertyChangedEventHandler handler = PropertyChanged;
    field = value;
    if (handler != null)
     handler(this,
             new PropertyChangedEventArgs(propertyName.Substring(4)));
    return true;
   }
 }
```

Nous transmettons donc, par référence, le champ à modifier, la valeur à assigner à ce même champ et la propriété, sous forme de `string`, dont nous retirons les 4 premiers caractères car la méthode que nous allons utiliser renvoie le nom de la propriété préfixé par les 4 caractères `set_` à supprimer. Pour terminer, il s'agit évidemment d'une méthode générique s'adaptant ainsi à tous les types de champs pouvant être rencontrés.

L'appel de cette nouvelle méthode se met dans les accesseurs en écriture sous la forme suivante.

```
public int ID
 {get { return _ID; }
  set { AssignerChamp<int>(ref  ID, value,
                           MethodBase.GetCurrentMethod().Name); }
 }
public string Nom
 {get { return _Nom; }
  set { AssignerChamp<string>(ref _Nom, value,
                              MethodBase.GetCurrentMethod().Name); }
 }
public string Prenom
 {get { return _Prenom; }
  set { AssignerChamp<string>(ref _Prenom, value,
                              MethodBase.GetCurrentMethod().Name); }
 }
```

Pour ne pas avoir de souci de compilation, il convient de ne pas oublier la reference à l'espace de noms où est définie la méthode `GetCurrentMethod()` via l'instruction

```
using System.Reflection;
```

Ceci permet de récupérer, sous forme de chaîne de caractères, le nom de l'accesseur préfixé par `set_` comme nous l'avons déjà expliqué.

5.4.2. Type de liaison

Passons maintenant à une analyse plus en détail de cette liaison entre l'interface utilisateur et le code behind.En effet, cette dernière peut s'effectuer de plusieurs manières, à savoir

- `OneWay` pour que les modifications effectuées sur les données présentes dans le code behind soient immédiatement répercutées dans l'interface utilisateur.
- `TwoWay` lie les données du code behind et de l'interface utilisateur dans les deux sens et, par conséquent, ce mode ajoute au mode `OneWay` la répercussion d'une modification depuis l'interface utilisateur aux données stockées dans le code behind.
- `OneTime` fonctionne comme le mode `OneWay` mais uniquement lors du chargement des données, permettant ainsi l'initialisation de l'interface utilisateur depuis le code behind.
- `OneWayToSource` est l'inverse du mode `OneWay`.
- `Default` reprend le mode défini par défaut du contrôle vise.

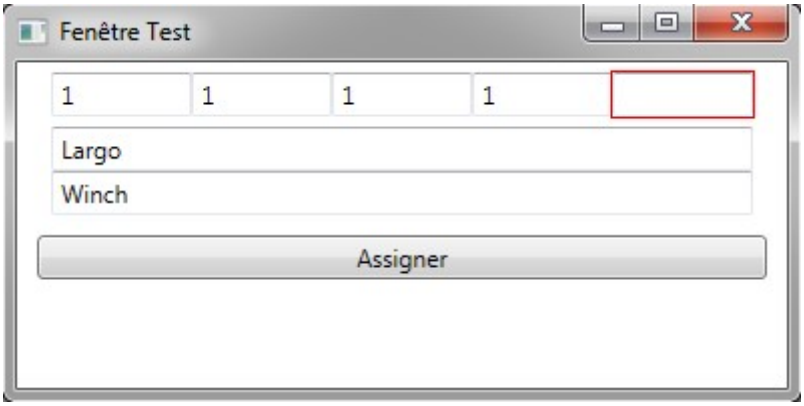
Nous illustrons ces concepts en modifiant notre interface utilisateur. Nous remplaçons les composants `TextBlock` par des composants de type `TextBox` utilisant les différents modes de propagation.

```
<StackPanel Name="Bibi">
 <StackPanel Margin="5" Orientation="Horizontal"
             HorizontalAlignment="Center">
  <TextBox Text="{Binding Path=ID}" Width="70" />
  <TextBox Text="{Binding Mode=TwoWay, Path=ID}" Width="70" />
  <TextBox Text="{Binding Mode=OneTime, Path=ID}" Width="70" />
  <TextBox Text="{Binding Mode=OneWay, Path=ID}" Width="70" />
  <TextBox Text="{Binding Mode=OneWayToSource, Path=ID}" Width="70" />
 </StackPanel>
 <TextBox Text="{Binding Path=Prenom}" Width="350" />
 <TextBox Text="{Binding Path=Nom}" Width="350" />
 <Button Margin="10" Content="Assigner" Click="btnAssigner_Click" />
</StackPanel>
```

Comme nous pouvons le constater, nous ajoutons également un bouton. Ce dernier permet d'affecter de nouvelles valeurs à l'objet `p` via le code behind.

```
private void btnAssigner_Click(object sender, RoutedEventArgs e)
 {p.ID = 24;
  p.Nom = "Haddock";
  p.Prenom = "Archibald";
 }
```

Le résultat est le suivant.



Il suffit alors de modifier la valeur de l'identifiant dans les différentes zones d'édition.

- Toute modification dans le code behind, causée par le bouton, est répercutée dans les première, deuxième et quatrième zones de texte, le mode par défaut étant, pour les composants de type `TextBox`, équivalent au mode `TwoWay`.
- Modifier l'identifiant des première, deuxième et cinquième zones de texte entraîne la mise à jour de la variable `p` définie dans le code behind

5.4.3. Liaison sur une collection d'objets

Evidemment, WPF ne s'est pas limité à définir des outils pour des données seules. Il permet également de manipuler aisément des ensembles de données.

Le principe est très simple. Comme précédemment, nous partons d'une classe de base implémentant l'interface `INotifyPropertyChanged` et utilisons une liste améliorée de type `ObservableCollection`. Comme annoncé, nous pouvons utiliser les méthodes d'ajout, de suppression, de parcours, … telles qu'elles existent dans la classe `List<>`. La seule contrainte, devenue habituelle, est de placer des accesseurs pour ce nouveau champ.

Il nous suffit de placer dans le code C# les instructions suivantes

```
private ObservableCollection<Personne> _lPers
         = new ObservableCollection<Personne>();
public ObservableCollection<Personne> lPers
{get { return _lPers; }
 set { _lPers = value; }
}
```

puis, dans le constructeur de la fenêtre,

```
lPers.Add(new Personne(1, "Winch", "Largo"));
lPers.Add(new Personne(2, "Ovronnaz", "Simon"));
lPers.Add(new Personne(3, "Blackman", "Catherine"));
```

Une fois le décor planté, il nous suffit de référencer cette resource dans le code XAML.

Nous commençons par ajouter dans la définition de la fenêtre une référence à la fenêtre elle-même de manière à accéder aux données contenues par cette dernière, en particulier `lPers`. Parallèlement à ce que nous avons vu en 0, nous écrivons

```
DataContext="{Binding RelativeSource={RelativeSource Self}}"
```

Ensuite, nous allons placer deux éléments dédicacés à la collection et trois à la gestion *individuelle* des données à partir des éléments *collectifs*.

Ainsi, nous avons

- un contrôle XAML de type `ListView` en liaison avec `lPers` présentant les données `Prenom` et `Nom` sous deux colonnes,
- un contrôle XAML de type `ListBox` en liaison avec `lPers` présentant les données `Nom`,
- un ensemble de trois contrôles de type `TextBox` destinés à reprendre les valeurs correspondant à l'élément sélectionné.

Le code peut se mettre sous la forme suivante

```
<StackPanel Margin="15">
 <StackPanel Orientation="Horizontal">
  <ListView Name="ListePersonnes" Height="100" Width="240"
            ItemsSource="{Binding Path=lPers}">
   <ListView.View>
    <GridView>
     <GridViewColumn Width="100" Header="Prénom"
                     DisplayMemberBinding="{Binding Prenom}"/>
     <GridViewColumn Width="100" Header="Nom"
                     DisplayMemberBinding="{Binding Nom}"/>
    </GridView>
   </ListView.View>
  </ListView>
  <ListBox Width="150" ItemsSource="{Binding Path=lPers}"
           DisplayMemberPath="Nom" />
 </StackPanel>
 <StackPanel Name="ZoneEdition" Margin="15" >
  <TextBox Name="tbID" Text="{Binding Path=SelectedItem.ID,
                                      ElementName=ListePersonnes}" />
  <TextBox Name="tbPrenom" Text="{Binding Path=SelectedItem.Prenom,
                                  ElementName=ListePersonnes}" />
  <TextBox Name="tbNom" Text="{Binding Path=SelectedItem.Nom,
                              ElementName=ListePersonnes}" />
 </StackPanel>
</StackPanel>
```

5.5. Les déclencheurs

Il est possible de définir des stratégies de mise à jour entre l'interface utilisateur et les données stockées en arrière plan. Cela passe par des déclencheurs (*triggers*) par l'intermédiaire de la propriété `Binding UpdateSourceTrigger`.

Nous disposons des choix associés suivants qui permettent de déterminer quand la mise à jour se déclenche effectivement.

- `LostFocus` lorsque le contrôle perd le focus au bénéfice d'un autre,
- `PropertyChanged` lorsque, dans le cas d'un `TextBox`, la valeur a été modifiée,
- `Explicit` pour définir un traitement personnalisé via `UpdateSource` dans le code C# pour, par exemple, soumettre la validation au clic sur un bouton.

Pour illustrer ce concept, il nous suffit de créer une fenêtre disposant de 3 composants de type `TextBox`, d'un de type `Label` et d'un dernier de type `Button` et faisant référence à un objet `p` de type `Personne` comme défini précédemment.

Nous pouvons écrire le code XAML suivant

```
<Window x:Class="SandBox.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Déclencheurs"
        xmlns:local="clr-namespace:SandBox">
 <StackPanel Margin="15">
  <Label>Déclencheur LostFocus</Label>
  <TextBox Name="tbLostFocus" Text="{Binding Path=Nom}" />
  <Label>Déclencheur PropertyChanged</Label>
  <TextBox Text="{Binding Path=Nom, UpdateSourceTrigger=PropertyChanged}"
           Name="tbPropertyChanged" />
  <Label>Déclencheur Explicit</Label>
  <TextBox Text="{Binding Path=Nom, UpdateSourceTrigger=Explicit}"
           Name="tbExplicit" />
  <Label Name="lblUpgrade">Vérification</Label>
  <TextBlock Text="{Binding Path=Nom}"/>
  <Button Content="Vérification 'Explicit'" Click="Button_Click"/>
 </StackPanel>
</Window>
```

Nous constatons alors, comme annoncé, que la modification se met en place

- à partir du premier `TextBox` lorsque nous *quittons* le contrôle,
- à partir du deuxième `TextBox` lors de toute modification, en direct,
- à partir du troisième `TextBox` lorsque nous cliquons sur le bouton moyennant le fait d'avoir placé le code suivant dans la méthode `Button_Click`.

```
BindingExpression be
 = tbExplicit.GetBindingExpression(TextBox.TextProperty);
be.UpdateSource();
```

Nous pouvons également revenir à l'exemple présenté en 5.4.3 qui présentait l'inconvénient d'une mise à jour immédiate des données, champ par champ. Nous savons maintenant que les déclencheurs par défaut y sont pour quelque chose et le type `Explicit` permet de confirmer, par l'intermédiaire d'un bouton, les modifications.

Il nous suffit d'ajouter à la suite des trois composants de type `TextBox` l'instruction XAML suivante.

```
<Button Name="btnConfirmer" Content="Confirmer"
        Click="btnConfirmer_Click"/>
```

Le code C# suivant rend alors effectives les modifications.

```
private void btnConfirmer_Click(object sender, RoutedEventArgs e)
{BindingExpression be = tbNom.GetBindingExpression(TextBox.TextProperty);
 be.UpdateSource();
 be = tbPrenom.GetBindingExpression(TextBox.TextProperty);
 be.UpdateSource();
}
```

ou, de manière plus *classique*,

```
private void btnConfirmer_Click(object sender, RoutedEventArgs e)
{for (int i = 0; i < lPers.Count; i++)
  if (int.Parse(tbID.Text) == lPers[i].ID)
  {lPers[i].Prenom = tbPrenom.Text;
   lPers[i].Nom = tbNom.Text;
   break;
  }
}
```

5.6. La validation des données

Puisqu'il est possible de mettre directement à jour les données, il est bon de prévoir des méthodes de vérification de ces dernières pour éviter quelques désagréables surprises.

Nous commençons par définir un style qui permet de mettre en évidence les problèmes par un encadrement en rouge et l'apparition d'une *bulle* explicative. Nous plaçons le code XAML suivant

```
<Window.Resources>
 <Style x:Key="sErreur" TargetType="{x:Type TextBox}">
  <Style.Triggers>
   <Trigger Property="Validation.HasError" Value="true">
    <Setter Property="BorderBrush" Value="Red" />
    <Setter Property="BorderThickness" Value="1" />
    <Setter Property="ToolTip"
            Value="{Binding RelativeSource={RelativeSource Self},
                    Path=(Validation.Errors)[0].ErrorContent}" />
   </Trigger>
  </Style.Triggers>
 </Style>
</Window.Resources>
```

Comme on peut le voir, le style est associé à un déclencheur qui permet d'activer ce style lorsqu'une erreur est rencontrée par l'intermédiaire de la propriété `Validation.HasError`. Nous récupérons le message d'erreur créé dans la bulle.

Nous pouvons maintenant passer au contrôle de validation en nous inspirant de l'exemple précédent avec le déclencheur de type `PropertyChanged`.

```
<StackPanel Margin="15">
 <Label>Déclencheur PropertyChanged</Label>
 <TextBox Name="tbPropertyChanged"
          Style="{StaticResource sErreur}"
          Text="{Binding Path=Nom,
                         UpdateSourceTrigger=PropertyChanged,
                         ValidatesOnDataErrors=True}" />
</StackPanel>
```

Il nous reste maintenant à passer au code C# où il suffit de faire hériter la classe `Personne` de l'interface `IDataErrorInfo` et de mettre en place ce qui requis par cette dernière, à savoir

```
public string Error
{get { return null; } }
public string this[string name]
{get
 {string result = null;
  switch (name)
  {case "Nom":
    if (Nom == string.Empty)
    result = "Chaîne vide non permise";
    break;
  }
  return result;
 }
}
```

La personnalisation consiste à définir, pour chaque champ de la classe `Personne`, les vérifications à mettre en place et les messages correspondant destinés à être repris, comme annoncé, dans la bulle. Dans ce cas-ci, nous exigeons que le nom ne soit pas une chaîne vide.

Pour constater les effets du code précédent, il suffit d'ajouter dans le constructeur de la fenêtre le code suivant puis de supprimer le nom qui s'affiche dans le contrôle.

```
Per = new Personne(1, "Winch", "Largo");
tbPropertyChanged.DataContext = Per;
```

5.7. Les paramètres d'application

Une autre manière de stocker des données dans le cadre des applications est de passer par le fichier de configuration de cette dernière, à savoir `app.config`.

Nous allons voir comment manipuler ce type de fichier. Tout d'abord, pour le créer, il suffit de faire un clic droit sur le projet à partir de l'explorateur de solutions et de choisir Propriétés (autre manière de faire, passer par le menu Projet, Propriétés de …). Une fois là, il suffit de choisir l'onglet Paramètres.

Par exemple, nous allons créer une variable d'environnement `CouleurFond` destinée à enregistrer la couleur de fond de la fenêtre. Nous commençons par remplir les colonnes proposées

- La première colonne permet de renseigner le nom de la variable d'environnement, `CouleurFond` dans le cas qui nous intéresse.
- La deuxième colonne exige la définition du type de la variable, à savoir `System.Drawing.Color`.
- La troisième colonne permet de spécifier la *portée* du paramètre, à savoir
  - Si on choisit Application, la propriété est définie en lecture seule et ne peut être modifiée par l'application.
  - Si le choix se porte sur Utilisateur, ce implique que l'application peut modifier en cours d'exécution la valeur sauvegardée et ainsi être le reflet des choix de l'utilisateur. Nous optons pour ce choix
- La quatrième et dernière colonne permet d'enregistrer la valeur par défaut. Nous utilisons les ressources de Visual Studio pour définir une couleur.

6. Annexes

6.1. Markup extension

Nous avons vu en 2.4 comment utiliser des extensions de balisage. Penchons-nous maintenant sur des exemples moins théoriques et, par conséquent, plus utiles.

Nous allons, par l'intermédiaire de ces extensions, voir comment

- Récupérer les paramètres de l'application,
- Afficher les éléments d'une énumération dans un ItemControl,
- Localisation d'une application avec des ressources incorporées

III-A. Binding sur les paramètres de l'application

Depuis Visual Studio 2005, une fonctionnalité très utile est à la disposition des développeurs : les paramètres d'application, qui permettent de persister les préférences de l'utilisateur dans un fichier qui sera automatiquement chargé au démarrage du programme. Visual Studio génère une classe nommée *NomDeLApplication.Properties.Settings*, qui permet d'accéder aux paramètres de façon typée, via des propriétés.

Une façon « naïve » d'utiliser ces paramètres est d'affecter manuellement leur valeur aux propriétés correspondantes lors de l'initialisation. Par exemple, si on a la position, la taille et l'état de la fenêtre comme paramètres d'application :

Sélectionnez

```
public Window1()
{
    InitializeComponent();
    this.Left = Properties.Settings.Default.Left;
    this.Top = Properties.Settings.Default.Top;
    this.Width = Properties.Settings.Default.Width;
    this.Height = Properties.Settings.Default.Height;
    this.WindowState = Properties.Settings.Default.WindowState;
}
```

Cette façon de faire présente plusieurs inconvénients : d'une part, ça fait pas mal de code à écrire, surtout si on a beaucoup de paramètres ; d'autre part, il faut en écrire autant pour sauvegarder les valeurs lorsqu'on quitte l'application.

Pour « automatiser » ces tâches d'initialisation/sauvegarde, il suffit d'utiliser un *Binding*. Pour cela, on définit comme *Path* le nom du paramètre, et comme *Source* l'objet *Properties.Settings.Default*. Il faut aussi indiquer que le binding est en mode *TwoWay*, pour que les changements effectués par l'utilisateur soient répercutés dans les paramètres. Au final, ça donne quelque chose comme ça :

Sélectionnez
```
<Window x:Class="SettingMarkupExtension.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:p="clr-namespace:SettingMarkupExtension.Properties"
        Title="Window1"
        Height="{Binding Source={x:Static p:Settings.Default}, Path=Height,
Mode=TwoWay}"
        Width="{Binding Source={x:Static p:Settings.Default}, Path=Width,
Mode=TwoWay}"
        Left="{Binding Source={x:Static p:Settings.Default}, Path=Left,
Mode=TwoWay}"
        Top="{Binding Source={x:Static p:Settings.Default}, Path=Top,
Mode=TwoWay}"
        WindowState="{Binding Source={x:Static p:Settings.Default},
Path=WindowState, Mode=TwoWay}">
```

Seulement voilà : vu la lourdeur de la syntaxe, on n'a pas vraiment envie d'écrire ça pour chaque propriété qui est liée à un paramètre. C'est long à écrire, peu intuitif, et ça rend le code peu lisible. C'est typiquement dans ce genre de cas qu'on a intérêt à créer une markup extension pour nous simplifier la tâche.

Nous allons donc créer une extension nommée *SettingBindingExtension*, qui sera chargée de lier une propriété à un paramètre. Première différence par rapport à l'exemple du chapitre précédent : au lieu d'hériter directement de *MarkupExtension*, nous allons profiter des fonctionnalités déjà implémentées par *Binding* en héritant de cette classe. En effet, ce qu'on souhaite faire est finalement un binding un peu plus spécialisé. Tout ce qui nous reste à faire est d'initialiser les propriétés *Source* et *Mode* :

Sélectionnez
```
using System.Windows.Data;

namespace SettingMarkupExtension
{
    public class SettingBindingExtension : Binding
```

```
    {
        public SettingBindingExtension()
        {
            Initialize();
        }

        public SettingBindingExtension(string path)
            : base(path)
        {
            Initialize();
        }

        private void Initialize()
        {
            this.Source = Properties.Settings.Default;
            this.Mode = BindingMode.TwoWay;
        }
    }
}
```

Notez qu'on a implémenté 2 constructeurs, qui correspondent à ceux de la classe *Binding*. De cette façon on peut choisir d'initialiser *Path* par le constructeur, ou explicitement. Voyons maintenant ce que donne notre extension à l'usage, pour la même fonctionnalité que le code précédent :

Sélectionnez

```
<Window x:Class="SettingMarkupExtension.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:my="clr-namespace:SettingMarkupExtension"
        Title="Window1"
        Height="{my:SettingBinding Height}"
        Width="{my:SettingBinding Width}"
        Left="{my:SettingBinding Left}"
        Top="{my:SettingBinding Top}"
        WindowState="{my:SettingBinding WindowState}">
```

C'est tout de même nettement plus pratique. si on regarde le temps qu'il a fallu pour écrire l'extension par rapport au temps qu'elle va nous faire gagner, c'est largement rentable... Et, cerise sur le gâteau : ça fonctionne sans problème dans le designer WPF, qui utilise les valeurs par défaut des paramètres.

Notez aussi que, comme on hérite de la classe *Binding*, on peut toujours utiliser les propriétés héritées de *Binding* : on peut par exemple définir une valeur par défaut avec *FallbackValue*, au cas où, pour une raison ou une autre, on ne peut pas récupérer la valeur du paramètre :

Sélectionnez

```
        Top="{my:SettingBinding Top, FallbackValue=50}"
```

Et enfin, petit détail à ne pas oublier pour que ça fonctionne : enregistrer les paramètres avant de quitter. on peut faire ça dans l'évènement *Exit* de l'application :

Sélectionnez
```
        private void Application_Exit(object sender, ExitEventArgs e)
        {
            SettingMarkupExtension.Properties.Settings.Default.Save();
        }
```

Il serait intéressant d'externaliser cette markup extension dans une librairie réutilisable, ce qui permettrait notamment de l'enregistrer dans le namespace par défaut de WPF, et donc de se passer du préfixe de namespace. En l'état actuel, ce n'est pas possible, à cause de la référence explicite à la classe *Settings* qui est spécifique à notre application. On peut bien sûr contourner cet inconvénient en utilisant la réflexion, mais c'est un peu plus complexe, surtout si on veut que ça fonctionne aussi dans le designer. Mais là on s'éloigne du cadre de cet article...

III-B. Afficher les valeurs d'une énumération dans un ItemsControl

Il est courant, dans un formulaire, de proposer à l'utilisateur de choisir entre plusieurs valeurs qui correspondent aux valeurs d'une énumération. Dans le code-behind, il est assez facile d'afficher la liste de ces valeurs dans une *ComboBox* (ou autre contrôle hérité de *ItemsControl*) :

Sélectionnez
```
            comboDayOfWeek.ItemsSource = Enum.GetValues(typeof(DayOfWeek));
```

Même si cette instruction est très simple, il serait plus naturel de définir la source de données de la ComboBox de façon déclarative, en XAML, puisque ce sont des données purement statiques. Comme c'est une question assez récurrente, on trouve souvent la solution suivante sur le net, qui utilise un *ObjectDataProvider* :

Sélectionnez
```
<Window x:Class="EnumValuesMarkupExtension.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="Window1" Height="300" Width="300">
    <Window.Resources>
        <ObjectDataProvider x:Key="odp" ObjectType="{x:Type sys:Enum}"
MethodName="GetValues">
            <ObjectDataProvider.MethodParameters>
                <x:Type TypeName="sys:DayOfWeek"/>
            </ObjectDataProvider.MethodParameters>
        </ObjectDataProvider>
    </Window.Resources>
    <Grid>
        <ComboBox ItemsSource="{Binding Source={StaticResource odp}}"
SelectedIndex="0"/>
```

```
    </Grid>
</Window>
```

Le problème, avec cette méthode, est qu'il faut déclarer un *ObjectDataProvider* à chaque fois qu'on veut se binder sur les valeurs d'une énumération, ce qui est un peu lourd pour une tâche aussi simple.

Vous commencez à me voir venir : pour simplifier tout ça, on va créer une markup extension. A ce stade, vous avez probablement compris le principe, mais voici quand même le code de cette extension :

Sélectionnez

```
using System;
using System.Windows.Markup;

namespace EnumValuesMarkupExtensions
{
    [MarkupExtensionReturnType(typeof(Array))]
    public class EnumValuesExtension : MarkupExtension
    {
        public EnumValuesExtension()
        {
        }

        public EnumValuesExtension(Type enumType)
        {
            this.EnumType = enumType;
        }

        [ConstructorArgument("enumType")]
        public Type EnumType { get; set; }

        public override object ProvideValue(IServiceProvider serviceProvider)
        {
            return Enum.GetValues(EnumType);
        }
    }
}
```

On a donc défini une propriété *EnumType*, qui correspond au type de l'énumération dont on veut obtenir les valeurs. Puisque *Enum.GetValues* renvoie un *Array*, on l'indique dans l'attribut *MarkupExtensionReturnType*. Et comme d'habitude, on définit un constructeur qui prend en paramètre la valeur de la propriété *EnumType*.

Pour obtenir le même résultat que précédemment en utilisant cette extension, il suffit d'écrire :

Sélectionnez

```
<Window x:Class="EnumValuesMarkupExtension.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
        xmlns:my="clr-namespace:EnumValuesMarkupExtension"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="Window1" Height="300" Width="300">
    <Grid>
        <ComboBox ItemsSource="{my:EnumValues sys:DayOfWeek}"
SelectedIndex="0" />
    </Grid>
</Window>
```

Ce qui est tout de même nettement plus rapide et intuitif que de déclarer un *ObjectDataProvider*...

7. Références

XAML Overview (WPF), http://msdn.microsoft.com/en-us/library/ms752059%28v=vs.110%29.aspx

http://www.wpf-tutorial.com/

Les markup extensions en WPF, Thomas Levesque, http://www.developpez.com, 8 janvier 2009

C# et .NET, Version 2, Gérard Leblanc, Eyrolles, 2006.

Les nouveautés du langage C# 3.0, James Ravaille, Association DotNet France.

Linq to XML, Association DotNet France.

Linq to XML, http://www.dreamincode.net.

The Complete Visual C# Programmer's Guide, Bulent Ozkir, John Schofield, Levent Camlibel, Mahesh Chand, Mike Gold, Saurabh Nandu, Shivani Maheshwari, Srinivasa Sivakamur, Microgold Press, 2002.

Tutorial C#, Robert-Michel di Scala, www.developpez.com.

Utilisation des expressions régulières en .Net, Louis-Guillaume Morand, 2005.

Visual Studio .NET in 21 days, Jason Beres, Sams Publishing, 2003.

Visual Studio .NET 2003, Aide en ligne (basée sur .NET Framework version 1.1.4322), 15 novembre 2002.

A Guided Tour of WPF – Part … (XAML), Josh Smith, 2007, www.codeproject.com.

WPF: A Beginner's Guide - Part … of n, Sacha Barber, 2008, www.codeproject.com.

8. Table des matières