



WPF

Introduction :

La méthodologie Modèle / Vue / Vue-Modèle est une variation du patron de conception MVC.

Dans cette méthodologie l'interface utilisateur ou la réalisation de la Vue est confiée à un designer.

Un designer est concerné par les aspects graphiques et non par l'écriture de code.

Le design est réalisé en langage déclaratif (le XAML pour ce cours)

Quelques définitions:

- Le **modèle** constitue la couche de données métier et n'est lié à aucune représentation graphique précise. Il est réalisé en code C# et ne repose pas sur l'utilisation de technologie propre à WPF. Le Modèle n'a aucune connaissance de l'interface graphique qui lui est associée.

Introduction :

La Vue est constituée des éléments graphiques : fenêtre, bouton, sélecteur, etc.

Elle prend en charge les raccourcis clavier et ce sont les contrôles eux-mêmes qui gèrent l'interaction avec les périphériques de capture (clavier / souris)

Pour l'utilisation et la présentation des données à l'utilisateur, *le **DataBinding*** est préconisé.

Dans des cas simples, la Vue peut être directement "bindée" sur le Modèle.: par exemple le modèle expose une propriété booléenne qui est bindée sur un *CheckBox* dans la Vue.

Pour des cas plus complexes, le Modèle peut ne pas être directement exploitable par la Vue. Cela peut être le cas lors de la réutilisation d'un Modèle déjà existant qui ne présente pas les propriétés attendues par la Vue.

Introduction :

La partie **Vue-Modèle** peut être vue comme un adaptateur entre la Vue et le Modèle. Elle réalise plusieurs tâches :

- adaptation des types issus du Modèle en des types exploitables par la vue via le *databinding*;
- exposition des commandes que la vue utilisera pour interagir avec le Modèle;

les classes Vue-Modèle n'ont pas besoin de connaître la Vue qui les exploite.

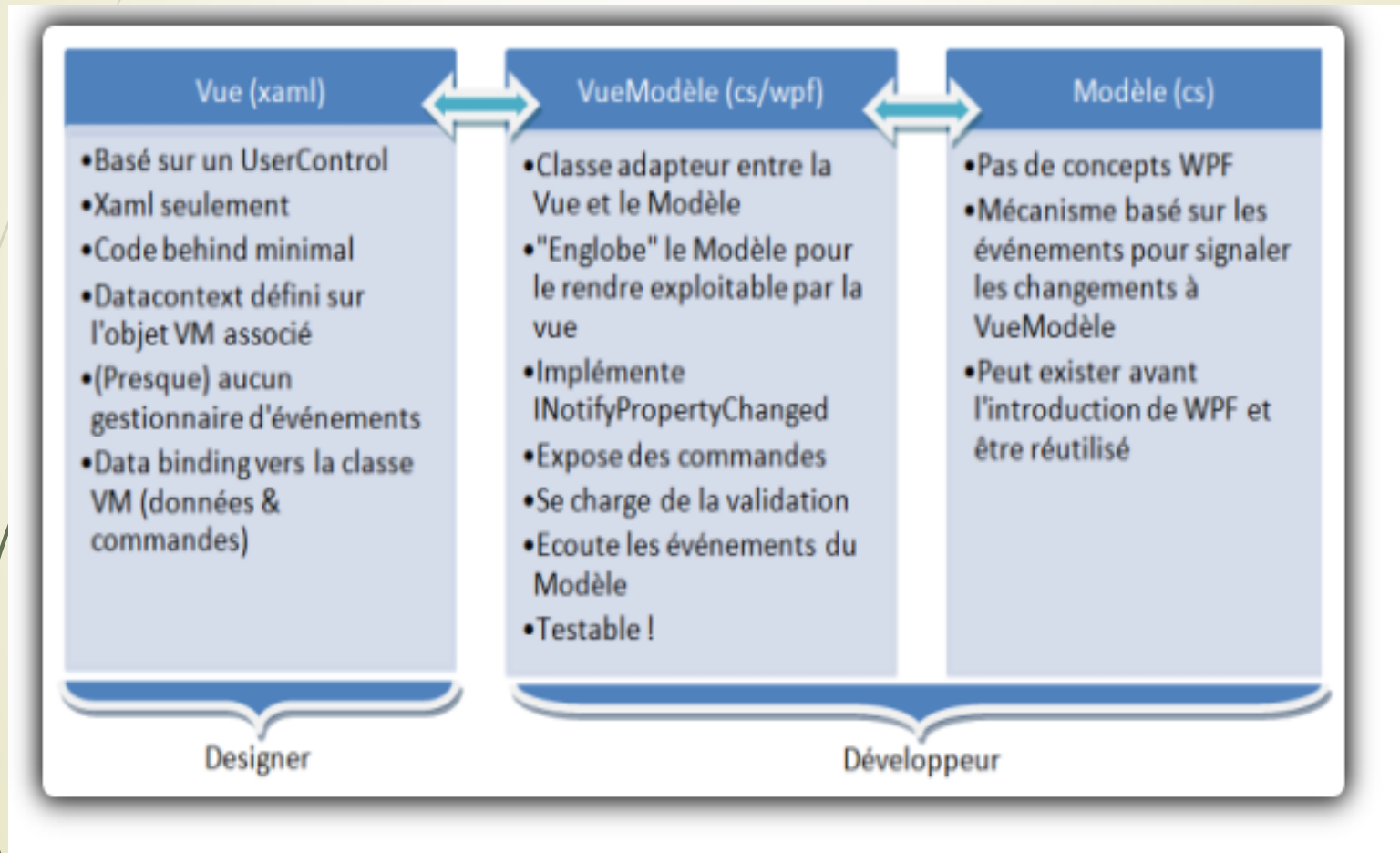
Les classes Vue-Modèle introduisent donc une abstraction entre la partie graphique (le XAML, réalisé par le designer) et le reste (les classes Vue-Modèle et Modèle réalisées par les développeurs).

Les fonctionnalités de WPF rendent l'utilisation de cette méthodologie complètement naturelle.

MVVM

Introduction :

En résumé:



Introduction :

WPF : Windows Presentation Foundation,

- WPF est un Framework d'IU (interface utilisateur) qui crée des applications clientes de bureau.
- La plateforme de développement WPF prend en charge un large éventail de fonctionnalités de développement d'applications, notamment un modèle d'application, des ressources, des contrôles, des graphiques, la disposition, la liaison de données, les documents et la sécurité.
- WPF fait partie de .NET
- .Net offre le choix entre les WindowsForms et depuis le Framework .Net 3.0 **WPF**.
- Les WindowsForms correspondent à la partie du Framework.Net responsable de l'interface utilisateur. Malheureusement elles posent un problème au niveau du travail collaboratif entre les développeurs et les designers.
- **WPF** facilite le design de l'IHM : les graphismes vectoriels, la transparence par pixels, les animations, l'adaptation de la résolution d'écran, le support des templates de data binding, etc.

Introduction : les avantages de WPF

➤ **L'utilisation GPU :**

Le GPU est le processeur graphique présent sur la carte graphique.

L'utilisation du GPU permet de déléguer une partie du travail du CPU au GPU qui va se charger de manipuler les données graphiques. Les GPU étant de plus en plus puissant et nos applications de plus en plus "jolies" et par conséquent lourdes...

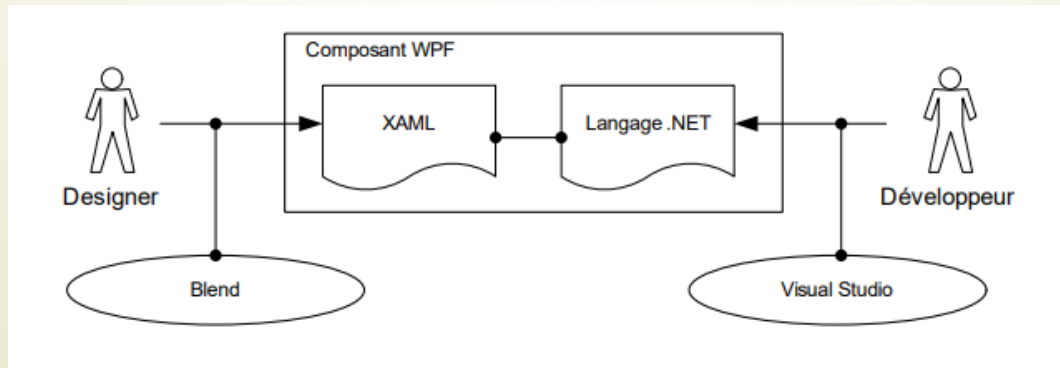
➤ **Séparation code/design :**

Au sein d'une équipe d'un même projet, il y a toujours des designers et des développeurs. Un problème majeur est posé : le designer devra avoir des compétences en développement, il va devoir connaître les objets de votre application et les fonctionnalités des WindowsForms : l'équipe va devoir adapter tout ça ...

Introduction : les avantages de WPF

WPF permet de résoudre ce problème : il permet de séparer en couches l'application :

- WPF va se charger de séparer le code designer du code *behind* (classe d'arrière-plan). C'est-à-dire que le designer va pouvoir travailler sur le design de l'application, via un langage commun basé sur le langage XML qui est le **XAML**.
- Quant au développeur de son côté via le code behind, va pouvoir travailler sur la couche métier. Cela va permettre une meilleure productivité et un support de l'application plus facile par la suite.



Introduction : les inconvénients de WPF

➤ **Manque d'interopérabilité**

Le principal problème de WPF reste l'interopérabilité de ce dernier. En effet on ne peut pas, dans l'état actuel des choses, faire du WPF sous linux, sous Mac OS ou d'autres systèmes... Le seul portage de .NET fait sur les autres systèmes reste mono, qui ne supporte actuellement pas le WPF.

➤ **Tout est à refaire**

La migration des applications WindowsForms actuelle reste complexe, le développeur qui souhaite migrer son application WindowsForms vers WPF, va devoir revoir et recréer toute son interface, il devra également revoir son architecture, notamment pour l'utilisation du Data Binding.

Le langage XAML

Le XAML (eXtensible Application Markup Language) est un langage déclaratif basé sur la syntaxe du XML.

Il permet grâce à des balises et des attributs de créer très facilement des objets. Pour cela, le compilateur XAML se charge de déclarer et définir des objets dynamiquement grâce aux balises (équivalent des classes) et aux attributs (équivalents aux propriétés) XAML.

Malgré sa syntaxe simple, le XAML permet de restituer des graphiques vectoriels, ou des modèles 3D aisément. Les possibilités graphiques sont donc infinies.

Il existe quelques règles élémentaires, issues de la syntaxe du XML, qu'il faudra respecter :

- Respecter la casse.
- Les balises ouvertes doivent être refermées sans se chevaucher.
- Chaque attribut doit obligatoirement avoir une valeur inscrite entre guillemets ou apostrophes.

Le langage XAML

Instancier nouvelle fenêtre

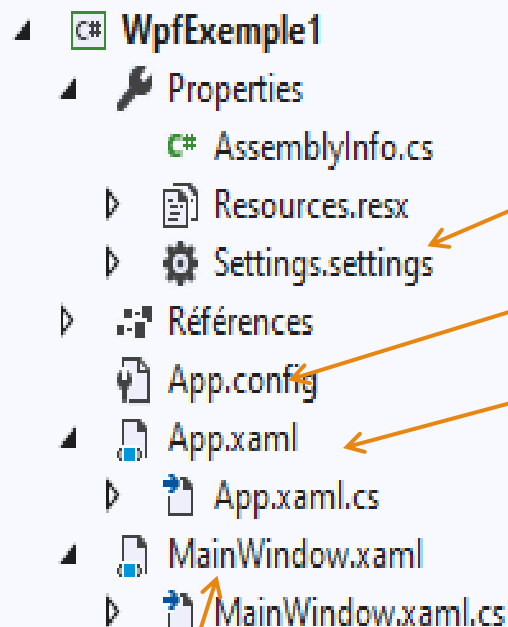
Exemple simple :

```
<Window x:Class="WpfExemple1.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:WpfExemple1"
  mc:Ignorable="d"
  Title="1er Exemple WPF" Height="450" Width="800">
  <Grid>
    <Button Margin="321,134,349,212" Content="Exemple Button"/>
    <TextBlock x:Name="txtBlk" Text=" Hello BUT 2A !"
      FontSize="26"
      FontFamily="MV Boli"
      HorizontalAlignment="Center"
      Foreground="Crimson" Margin="267,75,289,91" Width="236" >
    </TextBlock>
  </Grid>
</Window>
```

Éléments de la fenêtre

WPF

Structure d'une application WPF



Paramètres de l'application

Paramètres de configuration

Classe principale

The screenshot shows the App.xaml file in the Visual Studio Code editor. The code is as follows:

```
1 <Application x:Class="WpfExemple1.App"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4     xmlns:local="clr-namespace:WpfExemple1"
5     StartupUri="MainWindow.xaml">
6     <Application.Resources>
7     </Application.Resources>
8 </Application>
```

Fenêtre principale

The screenshot shows the App.xaml.cs file in the Visual Studio Code editor. The code is as follows:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;

namespace WpfExemple1
{
    /// <summary>
    /// Logique d'interaction pour App.xaml
    /// </summary>
    3 références
    public partial class App : Application
    {
    }
}
```

Structure d'une application WPF

- ✓ Le fichier **MainWindow.xaml** contient la description de la fenêtre principale de l'application,
- ✓ Le fichier de code-behind **MainWindows.xaml.cs** contient le code C# qui est associé au fichier **MainWindow.xaml**,
- ✓ Le fichier **App.config** contient les paramètres de configuration de l'application,
- ✓ Le fichier **AssemblyInfo.cs** contient les métadonnées décrivant l'assembly résultant de la compilation du projet : titre, version, informations de copyright... Ces informations sont décrites sous la forme d'attributs,
- ✓ Le fichier **Resources.resx** contient généralement des chaînes de caractères utilisables en plusieurs endroits de l'application, mais il peut aussi encapsuler des ressources de type binaire,
- ✓ Le fichier **Settings.settings** permet quant à lui de stocker et de récupérer des paramètres liés à l'application ou à un utilisateur de l'application : tailles de polices, couleurs ou emplacement des barres d'outils par exemple.

Passage de paramètres à une application WPF

1. Ajouter un constructeur à la classe application afin de lui ajouter la capture de l'événement Startup, celui-ci étant déclenché lors de l'appel de la méthode Run de l'application,
2. Récupérer les paramètres qui sont passés à la méthode associée à l'événement Startup au sein d'un objet de type *StartupEventArgs*, ces paramètres peuvent alors être stockés dans une donnée membre de l'application,
3. Récupérer les paramètres depuis la fenêtre : il est possible pour ceci de récupérer une référence à l'objet application à partir de la méthode statique *Application.Current* et par conséquent d'accéder aux paramètres stockés en données membres.

Exemple de passage de paramètres :

Dans l'exemple précédent, le contenu du *TextBloc* "Hello BUT 2A !" ne sera plus défini à l'avance mais passé en paramètre à l'application.

Passage de paramètres à une application WPF

Pour passer un paramètre à l'application nous allons modifier les fichiers *App.xaml.cs* et *MainWindow.xaml.cs* de la manière suivante :

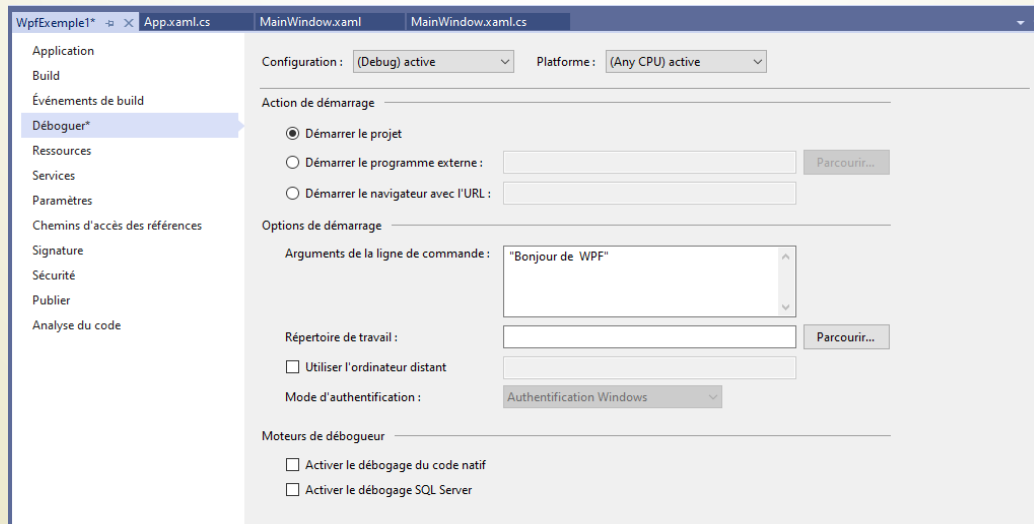

```
namespace WpfExemple1
{
    /// <summary>
    /// Logique d'interaction pour App.xaml
    /// </summary>
    public partial class App : Application
    {
        private string arg;
        public string Arg { get { return arg; } }
        public App()
        {
            this.Startup += App_Startup;
        }
        private void App_Startup(object sender, StartupEventArgs e)
        {
            if (e.Args.Length > 0)
                arg = e.Args[0];
            else
                arg = "";
        }
    }
}
```

The diagram illustrates the flow of parameters in a WPF application. It shows three files: *App.xaml*, *App.xaml.cs*, and *MainWindow.xaml.cs*. A blue arrow points from *App.xaml* to *App.xaml.cs*, indicating that the application logic is implemented in the code-behind file. A green arrow points from *App.xaml.cs* to *MainWindow.xaml.cs*, indicating that the application parameters are passed to the main window.

WPF

Passage de paramètres à une application WPF

```
namespace WpfExemple1
{
    /// <summary>
    /// Logique d'interaction pour MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            this.txtBlk.Text = ((App)Application.Current).Arg;
        }
    }
}
```



WPF : Présentations des settings

Les paramètres d'application permettent de stocker des informations sur l'application de manière dynamique. Ces informations ne doivent pas être incluses dans le code de l'application.

Ces informations peuvent être par exemple une chaîne de connexion à une base de données, les préférences utilisateur et d'autres informations dont on a besoin au moment de l'exécution.

Chaque paramètre d'application doit avoir un nom unique. Le nom peut être n'importe quelle combinaison de lettres, de chiffres ou de trait de soulignement. Il ne doit pas commencer par un nombre et il ne peut pas contenir d'espaces. Le nom est modifié par le biais de la propriété **Name**

Les paramètres d'application possèdent un **type** et une valeur. La valeur est définie avec la propriété **Value**.

Les paramètres d'application sont susceptibles de changer.

Pour modifier les paramètres d'application, il ne faut pas modifier le code source.

La persistance des données consiste à sauvegarder les paramètres que l'on veut maintenir dans leurs états quand on ferme l'application.

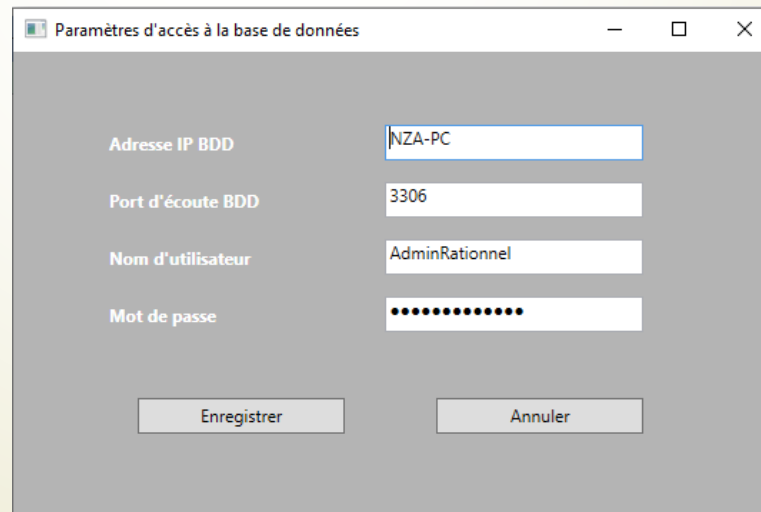
WPF : Présentations des settings

Exemple de mise en place des settings :

Par exemple pour une chaîne de connexion à une base de données, on peut être amené à changer de machine donc d'adresse IP ou alors changer de mot de passe utilisateur par sécurité;

On a donc besoin d'une fenêtre (par exemple pour une application WPF) pour les configurer et les sauvegarder pour des utilisations futures.

Exemple de fenêtre :



The screenshot shows a WPF window titled "Paramètres d'accès à la base de données". It contains four text input fields for configuration:

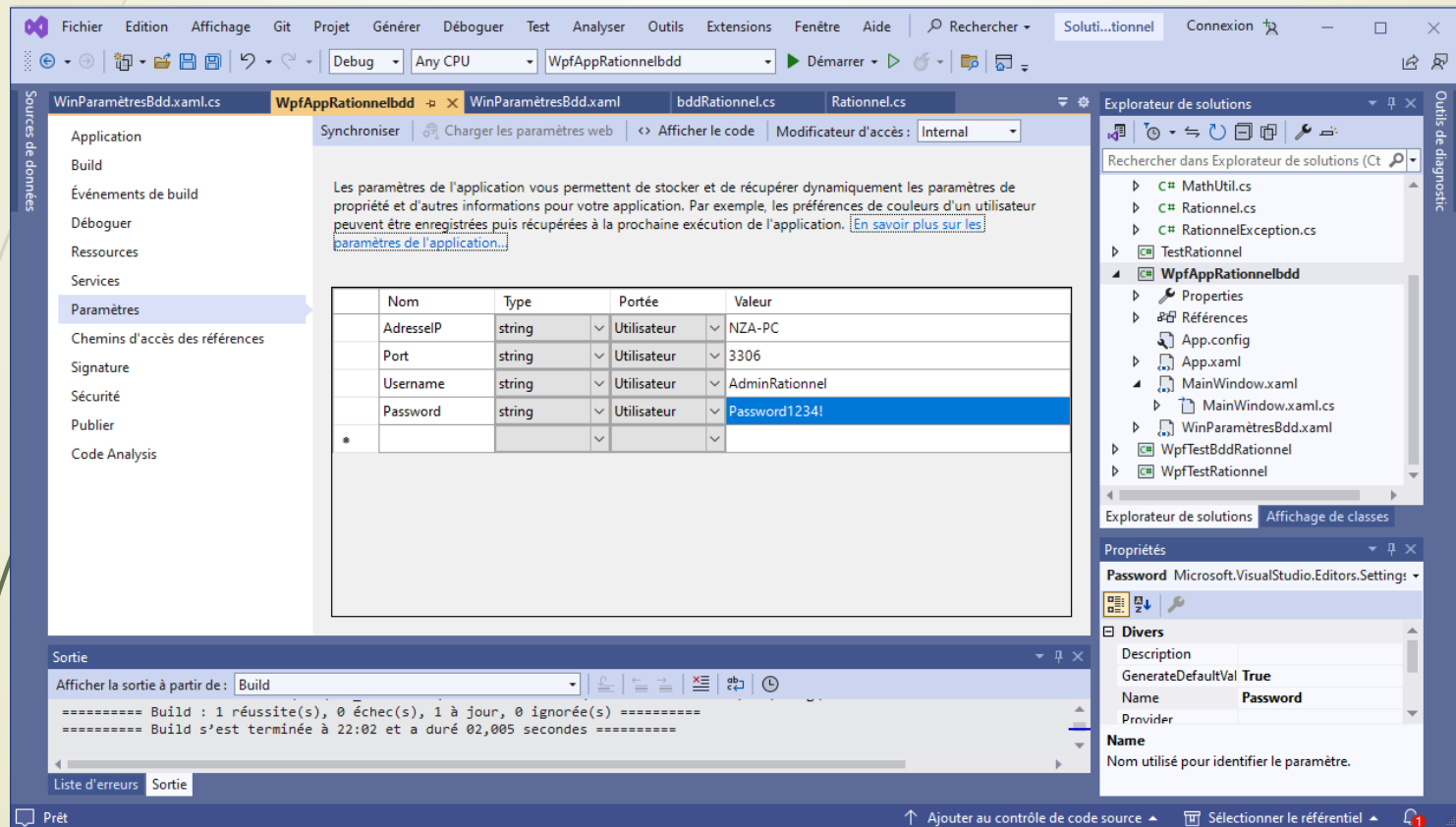
- Adresse IP BDD: NZA-PC
- Port d'écoute BDD: 3306
- Nom d'utilisateur: AdminRationnel
- Mot de passe: (masked with dots)

At the bottom, there are two buttons: "Enregistrer" (Save) and "Annuler" (Cancel).

WPF

WPF : Présentations des settings

Définition et initialisation des paramètres sous Visual Studio:



Les paramètres de l'application vous permettent de stocker et de récupérer dynamiquement les paramètres de propriété et d'autres informations pour votre application. Par exemple, les préférences de couleurs d'un utilisateur peuvent être enregistrées puis récupérées à la prochaine exécution de l'application. [En savoir plus sur les paramètres de l'application...](#)

Nom	Type	Portée	Valeur
AdresseIP	string	Utilisateur	NZA-PC
Port	string	Utilisateur	3306
Username	string	Utilisateur	AdminRationnel
Password	string	Utilisateur	Password1234!
*			

Sortie

Afficher la sortie à partir de: Build

```
===== Build : 1 réussite(s), 0 échec(s), 1 à jour, 0 ignorée(s) =====  
===== Build s'est terminée à 22:02 et a duré 02,005 secondes =====
```

Liste d'erreurs | Sortie

Propriétés

Microsoft.VisualStudio.Settings

Divers

Description

GenerateDefaultVal True

Name Password

Provider

Name

Nom utilisé pour identifier le paramètre.

WPF : Présentations des settings

Code source C# associé :

Instanciation et ouverture de la fenêtre des paramètres et sauvegarde des valeurs saisies dans le fichier des settings.

```
private void MenuItem_Click(object sender, RoutedEventArgs e)
{
    try
    {
        WinParamètresBdd paramBDD = new WinParamètresBdd();
        paramBDD.TxtBoxAdresseIP.Text = Properties.Settings.Default.AdresseIP;
        paramBDD.TxtBoxPort.Text = Properties.Settings.Default.Port;
        paramBDD.TxtBoxUsername.Text = Properties.Settings.Default.Username;
        paramBDD.Passwordbox.Password = Properties.Settings.Default.Password;

        if (paramBDD.ShowDialog() == true)
        {
            Properties.Settings.Default.AdresseIP = paramBDD.TxtBoxAdresseIP.Text;
            Properties.Settings.Default.Port = paramBDD.TxtBoxPort.Text;
            Properties.Settings.Default.Username = paramBDD.TxtBoxUsername.Text;
            Properties.Settings.Default.Password = paramBDD.Passwordbox.Password;
            Properties.Settings.Default.Save();
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "Erreur lors de l'ouverture des parametres !");
    }
}
```


WPF : Présentations des settings

Code des boutons de la fenêtre des paramètres :

```
private void BtnEnregistrer_Click(object sender, RoutedEventArgs e)
{
    this.DialogResult = true;
    this.Close();
}

private void BtnAnnuler_Click(object sender, RoutedEventArgs e)
{
    this.DialogResult = false;
    this.Close();
}
```

→ Test : *WpfAppRationnelbdd*

Les composants de WPF

WPF est fourni avec la plupart des composants d'interface utilisateur courants utilisés dans presque toutes les applications Windows, telles que [Button](#), [Label](#), [TextBox](#), [Menu](#), [ListView](#) et [ListBox](#).

WPF fournit une bibliothèque de contrôles qui contient des informations sur chacun des contrôles.

On peut ajouter un contrôle à une application à l'aide du langage XAML ou du code.

Il est fréquent d'être amené à changer l'apparence d'un contrôle pour ajuster l'aspect de l'application. Cela est possible grâce à l'une des opérations suivantes:

- Modifier la valeur d'une propriété du contrôle
- Créez un [Style](#) pour le contrôle.
- Créez un nouveau [ControlTemplate](#) pour le contrôle.

les composants de WPF

Principaux composants utilisés dans une fenêtre :

- **Grid** : Le contrôle Grid permet de positionner précisément ses éléments (bouton, listes labels ...) dans les lignes et les colonnes.
- **Button** : Le contrôle Button permet de réagir à l'entrée utilisateur à partir d'une souris, d'un clavier, d'un stylet ou d'un autre appareil d'entrée. Il déclenche un événement Click. Son contenu peut être simple tel que du texte ou complexe tel que des images.
- **Label** : Ce contrôle est utilisé pour fournir des informations dans l'interface utilisateur.
- **CheckBox** : On utilise une [CheckBox](#) dans l'interface utilisateur de l'application pour représenter les options qu'un utilisateur peut sélectionner.
- **ComboBox** : Le contrôle [ComboBox](#) permet de présenter aux utilisateurs une liste d'options. La liste est affichée et masquée au fur et à mesure que le contrôle se développe ou se réduit. Dans son état par défaut, la liste est réduite, affichant un seul choix. L'utilisateur clique sur un bouton pour afficher la liste complète des options.

les composants de WPF

- **ListBox** : un contrôle [ListBox](#) fournit aux utilisateurs une liste d'éléments sélectionnables.
- **ListBoxItem** : Représente un élément sélectionnable dans un ListBox.
- **ListView** : un contrôle [ListView](#) fournit l'infrastructure pour afficher un ensemble d'éléments de données dans différentes dispositions ou vues. Le contrôle ListView est un [ItemsControl](#), cela signifie qu'il peut contenir une collection d'objets de n'importe quel type (par exemple chaîne, image ou panneau). Le contrôle ListView présente les données selon son mode d'affichage, qui est spécifié par la propriété [View](#).
WPF fournit un mode d'affichage [GridView](#) qui partitionne le contenu du contrôle ListView en colonnes.
GridView fournit des propriétés et des méthodes pour spécifier le style du contenu des colonnes.
ListView dérive de ListBox. En règle générale, ses éléments sont membres d'une collection de données et sont représentés en tant qu'objets [ListViewItem](#) . ListViewItem est un [ContentControl](#) et ne peut contenir qu'un seul élément enfant. Toutefois, cet élément enfant peut être n'importe quel élément visuel.

les composants de WPF

- **TextBox** : Représente un contrôle qui peut être utilisé pour afficher ou éditer du texte non formaté.
- **TextBlock** : Fournit un contrôle pour l'affichage de petites quantités de contenu de flux.
- **StackPanel** : Ce contrôle permet de disposer des éléments enfants sur une seule ligne orientée horizontalement ou verticalement.
- **Menu** : Représente un contrôle de menu Windows qui permet d'organiser de façon hiérarchique des éléments associés à des commandes et à des gestionnaires d'événements.
- **MenuItem** : Représente un élément sélectionnable dans un Menu.
- **ContextMenu** : Représente un menu contextuel qui permet à un contrôle de proposer des fonctionnalités propres à son contexte.
- **ProgressBar** : Indique la progression d'une opération.
- **RadioButton** : Représente un bouton qui peut être sélectionné mais pas désélectionné par un utilisateur. On peut définir la propriété [IsChecked](#) d'une case d'option [RadioButton](#) en cliquant sur celle-ci, on ne peut la désélectionner que par programme.

les composants de WPF

- **GroupBox** : Représente un contrôle qui crée un conteneur avec bordure et entête pour le contenu de l'interface utilisateur.
- **Image** : Le contrôle [Image](#) est utilisé pour afficher des images bitmap dans les applications WPF.
- **RichTextBox** : Le contrôle [RichTextBox](#) permet d'afficher ou de modifier du contenu de flux, notamment des paragraphes, des images, des tableaux, etc.. C'est un contrôle d'édition avec prise en charge intégrée des fonctionnalités telles que couper et coller.
- **ToolTip** : Le contrôle [ToolTip](#) est une info-bulle, c'est une petite fenêtre contextuelle indépendante, qui s'affiche lorsqu'un utilisateur suspend le pointeur de la souris sur un élément.
- **PasswordBox** : Le contrôle [PasswordBox](#) représente un contrôle conçu pour la saisie et la gestion des mots de passe.
- **DataGrid** : Le contrôle [DataGrid](#) permet d'afficher des données dans une grille personnalisable.
- **DatePicker** : Le contrôle [DatePicker](#) permet à l'utilisateur de sélectionner une date.

Propriété *DisplayMemberPath*

Les contrôles ListView ou ListBox étant des ItemsControl, il est possible d'utiliser la propriété ***DisplayMemberPath***.

Cette propriété permet de définir ou d'obtenir le chemin d'une valeur de la propriété de l'objet source qui sert dans la représentation visuelle de cet objet.

DisplayMemberPath est une propriété utilisée dans la programmation d'applications avec des interfaces utilisateurs comme celles de WPF.

Elle permet de spécifier quel champ ou quelle propriété d'un objet de données doit être affiché dans un contrôle de liste.

C'est ce qui définit ce que nous voyons dans une liste déroulante par exemple.

WPF : DataTemplate

Les *DataTemplate* sont des modèles qui permettent de contrôler le formatage des données d'une liste par exemple.

Un *DataTemplate* est donc utilisé pour fournir une structure visuelle aux données sous-jacentes.

Un *DataTemplate* contient des expressions Binding standard, liées aux propriétés de son DataContext (objet métier / modèle de vue). Il permet donc de décrire comment afficher les données.

Exemple: modéliser
une liste de fournisseurs

```
public class Fournisseur {
    private int nf;
    public int NF { get { return nf; } set { nf = value; } }
    private string nomF;
    public string NomF { get { return nomF; } set { nomF = value; } }
    private string status;
    public string Status { get { return status; } set { status = value; } }
    private string ville;
    public string Ville { get { return ville; } set { ville = value; } }
    private string urlLogo;
    public string UrlLogo { get { return urlLogo; } set { urlLogo = value; } }
    public Fournisseur(int numéro, string nom, string status, string ville, string url_logo) {
        this.nf = numéro;
        this.nomF = nom;
        this.status = status;
        this.ville = ville;
        this.urlLogo = url_logo;
    }
}
```

WPF : DataTemplate : Liste de fournisseurs

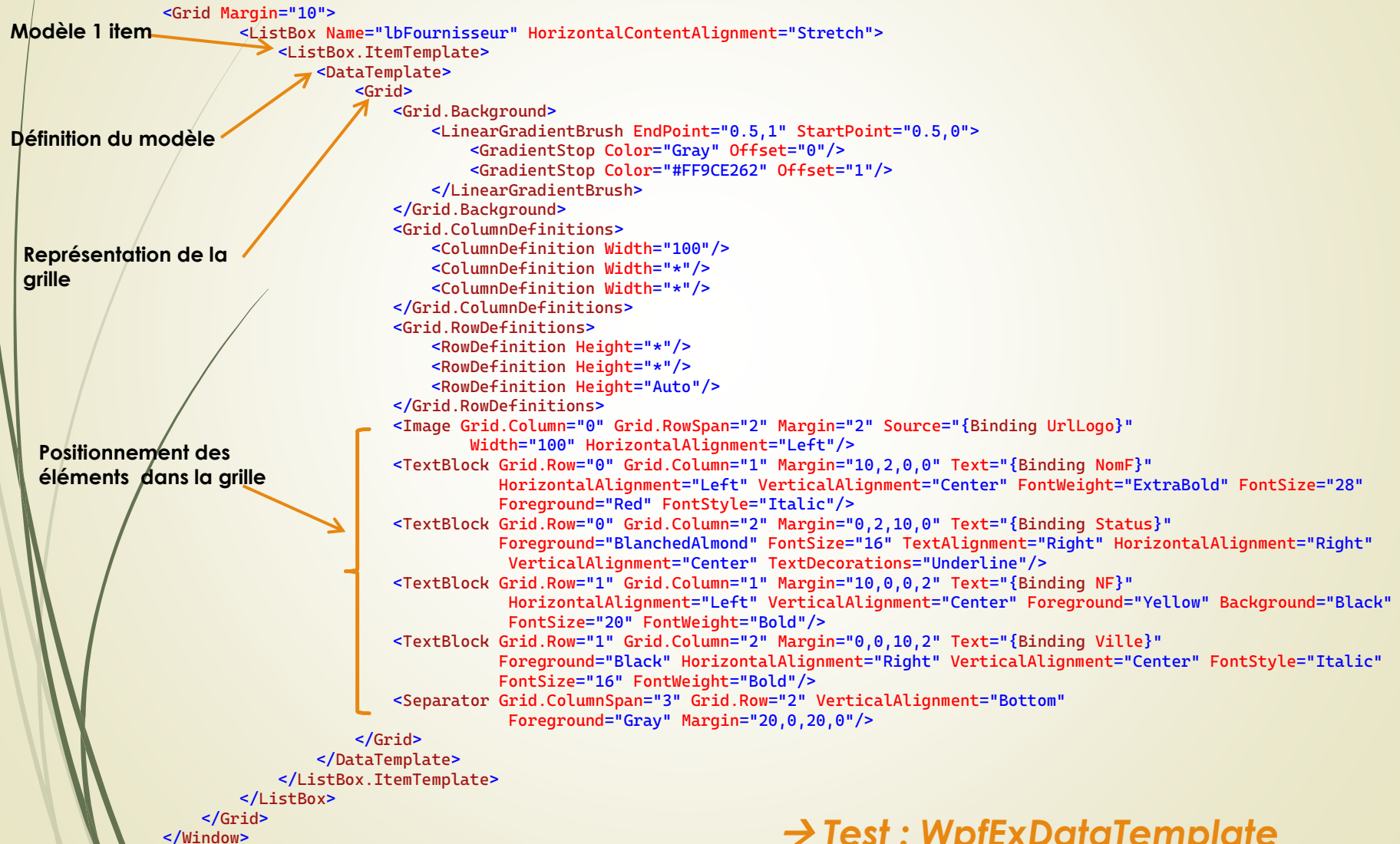
On souhaite afficher la liste des fournisseurs de la manière suivante :



Chaque item est représenté sur 2 lignes et deux colonnes.

L'espace du contrôle Grid est partagé de telle sorte à placer le nom et le numéro du fournisseur à gauche et le status ainsi que la ville à droite

WPF : DataTemplate: Liste de fournisseurs



WPF : DataTemplate

Il est possible de définir le modèle dans les ressources et de l'appeler par la suite.

```
<Window.Resources>
  <DataTemplate x:Key="TemplateFournisseur">
    <Grid>
      <Grid.Background ...>
      <Grid.ColumnDefinitions ...>
      <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>
      <Image Grid.Column="0" Grid.RowSpan="2" Margin="2" Source="{Binding UrlLogo}" ... />
      <TextBlock Grid.Row="0" Grid.Column="1" Margin="10,2,0,0" Text="{Binding NomF}" ... />
      <TextBlock Grid.Row="0" Grid.Column="2" Margin="0,2,10,0" Text="{Binding Status}" ... />
      <TextBlock Grid.Row="1" Grid.Column="1" Margin="10,0,0,2" Text="{Binding NF}" ... />
      <TextBlock Grid.Row="1" Grid.Column="2" Margin="0,0,10,2" Text="{Binding Ville}" ... />
      <Separator Grid.ColumnSpan="3" Grid.Row="2" VerticalAlignment="Bottom" ... />
    </Grid>
  </DataTemplate>
</Window.Resources>
```

```
<ListBox Name="lbFournisseur" HorizontalContentAlignment="Stretch"
  ItemTemplate="{StaticResource TemplateFournisseur }">
```

```
</ListBox>
```

→ Test : WpfExDataTemplate2

WPF : DataGrid

Le contrôle [DataGrid](#) permet d'afficher et de modifier des données. Ces données peuvent avoir des sources différentes:

- Une base de données ,
- Une requête LINQ,
- Une collection observable,
- Ou une autre source de données pouvant être liée.

Les colonnes d'un DataGrid peuvent contenir du texte, des contrôles, ou tout autre contenu WPF. (images, des boutons, ... tout ce que contient un modèle).

Il est possible d'utiliser un [DataGridTemplateColumn](#) pour afficher les données définies dans un modèle.

Il est possible d'appliquer toutes les fonctionnalités de style et de création de modèles d'autres contrôles WPF au contrôle DataGrid.

De même le contrôle DataGrid inclut les fonctionnalités par défaut et personnalisables pour la modification, le tri et la validation de données.

DataGrid permet aux utilisateurs d'interagir avec le contrôle à l'aide du clavier et de la souris.

WPF : DataGrid

Le contrôle DataGrid est personnalisable, on peut ajouter, trier, grouper et filtrer des données. On peut également effectuer une validation des données au niveau d'une cellule ou d'une ligne.

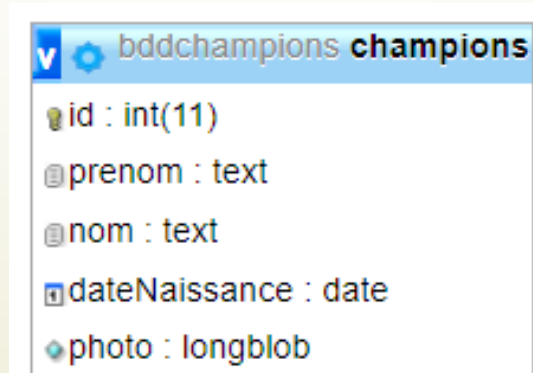
La validation au niveau d'une cellule, permet de valider les propriétés individuelles d'un objet de données lié lorsqu'un utilisateur met à jour une valeur.

La validation au niveau d'une ligne permet de valider l'intégralité des objets de données lorsqu'un utilisateur valide les modifications apportées à une ligne.

Exemple simple d'utilisation d'un DataGrid :

Affichage d'une liste de champions de tennis dans un DataGrid dont certaines propriétés sont personnalisées.






Chaque ligne du DataGrid est remplie par data-Binding sur une table ***champions*** d'une base de données ***bddchampions***



WPF : DataGridView champions


Fenêtre de l'application Champion:

liste des champions de tennis

Id	Nom	Prénom	Photo	date de naissance	
1	FEDERER	rodger		8/8/1981	
2	NADAL	raphael		6/3/1986	
3	DJOKOVIC	novak		5/22/1987	
4	ZVEREV	alexander		4/20/1997	
5	MONFILS	gael		9/1/1986	

Nom

Prénom

Choisir photo 

Date de naissan

Ajouter

Datagrid " bindé " sur la table *champions*

WPF : définition XAML du DataGrid

```
<DataGrid x:Name="DataGridChampions" AutoGenerateColumns="False" Margin="43,5,15,34" Grid.RowSpan="6" Background="#FF8EB4B" Grid.Row="1">
  <DataGrid.ContextMenu>
    <ContextMenu Name="ContextMenuListChampion" >
      <MenuItem Name="MenuContextSupprimerChampion" Header="Supprimer un champion" Click="MenuContextSupprimerChampion_Click" />
    </ContextMenu>
  </DataGrid.ContextMenu>
  <DataGrid.Columns>
    <DataGridTemplateColumn Header="Id">
      <DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
          <Label Content="{Binding Id}" />
        </DataTemplate>
      </DataGridTemplateColumn.CellTemplate>
    </DataGridTemplateColumn>
    <DataGridTemplateColumn Header="Nom">
      <DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
          <Label Content="{Binding Nom}" />
        </DataTemplate>
      </DataGridTemplateColumn.CellTemplate>
    </DataGridTemplateColumn>
    <DataGridTemplateColumn Header="Prénom">
      <DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
          <Label Content="{Binding Prenom}" />
        </DataTemplate>
      </DataGridTemplateColumn.CellTemplate>
    </DataGridTemplateColumn>
    <DataGridTemplateColumn Header="Photo">
      <DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
          <Image Source="{Binding Photo}" Width="100" />
        </DataTemplate>
      </DataGridTemplateColumn.CellTemplate>
    </DataGridTemplateColumn>
    <DataGridTemplateColumn Header="date de naissance">
      <DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
          <Label Content="{Binding DateNaissance}" />
        </DataTemplate>
      </DataGridTemplateColumn.CellTemplate>
    </DataGridTemplateColumn>
  </DataGrid.Columns>
</DataGrid>
```

Propriété initialisée à false pour pouvoir personnaliser le DataGrid

ContextMenu pour la suppression d'un champion

Personnalisation du contenu de chaque ligne en utilisant *DataGridTemplateColumn*

WPF : Gestion de la photo : Contrôle Image

Le contrôle [Image](#) est utilisé pour afficher des images bitmap dans les applications WPF.

Les types d'images supportés sont les suivants : .bmp, .gif, .ico, .jpg, .png, .wdp et .tiff.

La propriété `Source` définit l'image source du contrôle qui est de type [ImageSource](#). Or la photo d'un champion est de type *Byte[]* (tableau d'octets permettant de stocker les différents pixels de la photo). Il faut donc convertir le tableau de `Byte` en *ImageSource* et vice versa.

ImageSource est une classe abstraite donc non directement instanciable. Elle possède des définitions de méthodes qu'elle impose aux classes dérivées de coder pour gérer les images.

Elle possède trois classes dérivées qui sont également abstraites ([D3DImage](#), [DrawingImage](#), [BitmapSource](#))

La classe *BitmapSource*, qui est également abstraite possède des classes qui ne sont pas abstraites et qui permettent de faire les conversions qui sont nécessaires.

La classe [BitmapImage](#) Fournit une *BitmapSource* spécialisée qui est optimisée pour le chargement des images avec XAML

WPF : Exemple de méthodes de conversion

```
public BitmapImage ConvertByteArrayToBitmapImage(Byte[] imageByte)
{
    try
    {
        BitmapImage bi = new BitmapImage();
        bi.BeginInit();
        bi.StreamSource = new MemoryStream(imageByte);
        bi.EndInit();
        return bi;
    }
    catch { throw; }
}
```

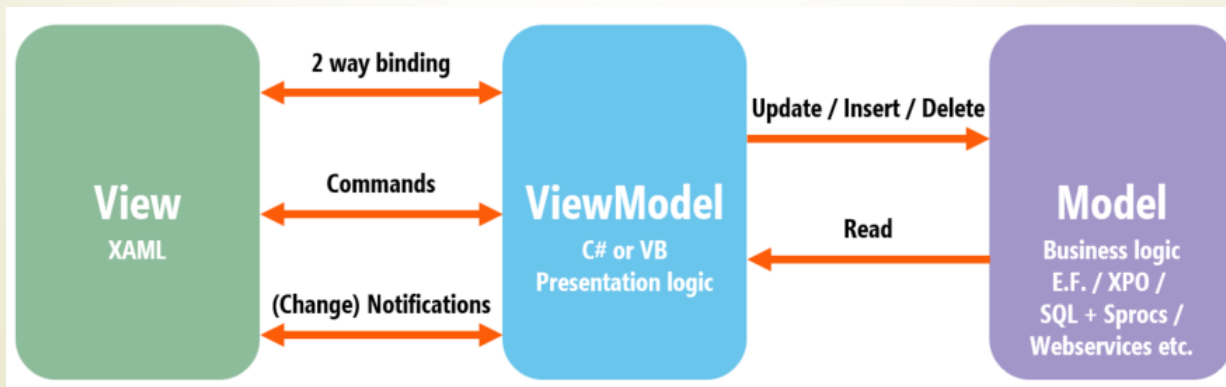
```
public byte[] ConvertImageToByteArray(System.Drawing.Image photo)
{
    try
    {
        using (var ms = new MemoryStream())
        {
            photo.Save(ms, photo.RawFormat);
            return ms.ToArray();
        }
    }
    catch { throw; }
}
```

→ Test : *WpfExDataGrid*

WPF : Le DataBinding

Définition:

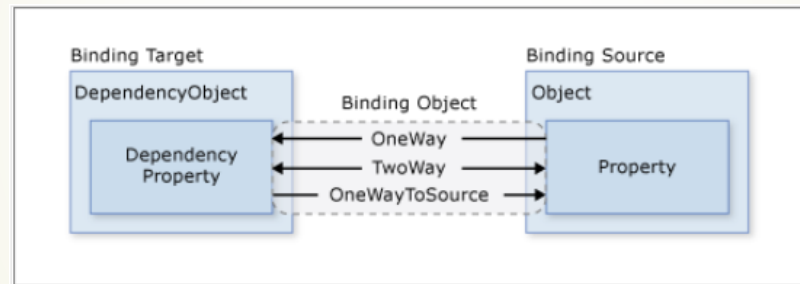
- Le DataBinding ou liaison de données est un mécanisme dans les applications WPF qui fournit un moyen simple et facile aux applications Windows Runtime d'afficher et d'interagir avec les données.
- C'est le processus qui établit une connexion entre l'interface utilisateur d'application et les données affichées.



WPF : Le DataBinding

La liaison de données dans WPF présente plusieurs avantages par rapport aux modèles traditionnels :

- Une représentation flexible des données dans l'interface utilisateur,
- Une séparation nette de la logique métier et de l'interface utilisateur,



chaque liaison a quatre composants :

- Un objet cible de liaison.
- Une propriété cible.
- Une source de liaison.
- Un chemin d'accès à la valeur de la source de liaison à utiliser.

WPF : Le *DataBinding*

Par exemple, si on veut lier le contenu d'un *TextBox* à la propriété *Employe.Nom*, d'un objet *Employe* on doit configurer la liaison selon le tableau suivant :

Paramètre	Valeur
Cible	TextBox
Propriété cible	Text
Objet source	Employe
Valeur de l'objet source	Nom

Remarques :

- ✓ Les sources de liaisons ne sont pas limitées aux objets .Net personnalisés. On peut aussi lier les contrôles à des données XML, des bases de données ou des collections d'objets.
- ✓ Le *DataBinding* permet de réaliser des conversions ou des contrôles de validation sur les valeurs liées.

WPF : Le contexte des données

Pour lier des données, il faut les déclarer sur les éléments XAML.

La propriété ***DataContext*** permet de définir l'objet source de données pour l'élément sélectionné.

Le contexte de données utilisé pour résoudre la liaison, est hérité du parent, sauf si celle-ci est définie explicitement sur l'objet.

Si la propriété *DataContext* de l'objet hébergeant la liaison n'est pas définie, la propriété *DataContext* de l'élément parent est vérifiée, et ainsi de suite, jusqu'à la racine de l'arborescence d'objets XAML.

Lorsque la propriété *DataContext* change, toutes les liaisons susceptibles d'être affectées par le contexte de données sont réévaluées.

WPF : Direction du flux de données

Les types de la liaison de données :

➤ **OneWay data binding :**

La liaison [OneWay](#) entraîne des modifications de la propriété source pour automatiquement mettre à jour la propriété cible, mais les modifications apportées à la propriété cible ne sont pas à nouveau propagées vers la propriété source. Ce type de liaison est approprié si le contrôle lié est implicitement en lecture seule. S'il n'est pas nécessaire de surveiller les modifications de la propriété cible, l'utilisation du mode de liaison OneWay permet d'éviter la surcharge du mode de liaison [TwoWay](#).

➤ **TwoWay data binding :**

La liaison TwoWay entraîne des modifications à la propriété source ou à la propriété cible pour mettre à jour automatiquement l'autre. Ce type de liaison convient aux formulaires modifiables.

La plupart des propriétés sont par défaut liées à OneWay mais certaines propriétés de dépendance (généralement des propriétés de contrôles modifiables par l'utilisateur, telles que [TextBox.Text](#) et [checkBox.IsChecked](#), par défaut, sont liées à TwoWay).

WPF : Direction du flux de données

- **OneWayToSource** : est l'inverse de la liaison *OneWay* ; il met à jour la propriété source quand la propriété cible change.

Un exemple de scénario : si on doit seulement réévaluer la valeur source à partir de l'interface utilisateur.

- **OneTime** : La liaison *OneTime* provoque l'initialisation par la propriété source de la propriété cible, mais elle ne propage pas les modifications ultérieures. Si le contexte de données change ou si l'objet dans le contexte des données change, la modification n'est pas répercutée dans la propriété cible.

WPF : La classe Binding

Les principales propriétés de cette classe sont :

- ✓ **Source** : qui fait référence aux données source d'une liaison. Par défaut la source est fournie par le "DataContext". Si on définit cette propriété sur une valeur non nulle, alors ce sera l'objet spécifié qui sera utilisé comme source de "data binding" pour l'élément courant.
- ✓ **Path** : qui est utilisé pour indiquer quelle propriété de l'objet source sera utilisée pour fournir les données liées. C'est une propriété de type **PropertyPath**, qui permet de prendre en charge une gamme complexe d'expressions de chemin d'accès.
- ✓ **ElementName** : qui peut être utilisé comme une alternative à la propriété source décrite précédemment. Utilisée dans le cas où la propriété Name ou l'attribut XAML `x:Name` correspond à l'élément de donnée à lier.
- ✓ **Converter** : propriété qui référence un objet qui doit implémenter l'interface *IValueConverter* ayant pour vocation de réaliser une conversion sur les données liées.

WPF : La classe Binding

- ✓ **Mode** : énumération qui permet de spécifier le sens de mise à jour de la donnée liée :
 - *OneWay* : lecture seule, l'élément graphique cible reflète la donnée source,
 - *OneWayToSource* : l'inverse du mode précédent, la source est affectée selon l'élément graphique,
 - *TwoWay* : liaison bidirectionnelle, en lecture-écriture,
 - *OneTime* : lecture seule, l'élément graphique cible reflète la donnée source initiale, mais pas ses modifications,
- ✓ **ValidationRules** : dans les modes "TwoWay" ou "OneWayToSource", une validation des données peut être réalisée à l'aide d'objets spécialisés à cet effet. Ces objets de validation sont alors référencés par la propriété *ValidationRules* qui est une collection d'objets *ValidationRule*,
- ✓ **UpdateSourceTrigger** : toujours dans les modes "TwoWay" ou "OneWayToSource", cette propriété permet de signaler l'événement origine de la répercussion sur l'objet source.

Pour détecter les modifications sources (applicables aux liaisons *OneWay* et *TwoWay*), la source doit implémenter un mécanisme de notification de modification de propriété approprié tel que [INotifyPropertyChanged](#).

WPF : La classe Binding

Exemple simple :

Le texte d'un label est initialisé avec le contenu d'un TextBox sans écrire aucune ligne de code.

- L'élément cible est le *label*,
- la propriété cible est *Content*,
- la donnée source est l'élément *TextBox*
- la propriété source est la propriété *Text* du *TextBox*
- le mode de liaison est le mode *OneWay*.

WPF

WPF : La classe Binding

```
<Window x:Class="WpfEx1Binding.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:WpfEx1Binding"
  mc:Ignorable="d"
  Title="MainWindow" Height="450" Width="800">
  <Grid>
    <Label x:Name="lblTexte" Content="{Binding ElementName=txtTexte,
      Path=Text,Mode=OneWay}" HorizontalAlignment="Left" Margin="38,107,0,0"
      VerticalAlignment="Top" Width="150" Height="30" Background="{DynamicResource
        {x:Static SystemColors.ControlLightBrushKey}}"/>
    <TextBox x:Name="txtTexte" HorizontalAlignment="Left" Height="30"
      Margin="230,107,0,0" TextWrapping="Wrap" VerticalAlignment="Top" Width="444"/>
    <Button x:Name="btnAfficher" Content="Afficher" HorizontalAlignment="Left"
      Margin="320,191,0,0" VerticalAlignment="Top" Width="75"
      Click="btnAfficher_Click"/>
  </Grid>
</Window>
```

→ Test : *WpfEx1Binding*

WPF : La classe Binding

Exemple TwoWay :

Réaliser une liaison bidirectionnelle entre un contrôle *Slider* et un *TextBox* :

```
<Grid>
<Slider x:Name="NbSlider" HorizontalAlignment="Left" Margin="125,165,0,0"
    VerticalAlignment="Top" Height="52" Width="275" Maximum="50" Value="25"
    Background="#FFC5D6CF"/>
<TextBox x:Name="txtNb" HorizontalAlignment="Left" Height="52"
    Margin="550,165,0,0" TextWrapping="Wrap" Text="{Binding
    ElementName=NbSlider,Path=Value,Mode=TwoWay}" VerticalAlignment="Top"
    Width="148" Background="#FF66C389" FontWeight="Bold"
    OpacityMask="#FFE6D0D0" TextAlignment="Center" FontSize="24"/>
</Grid>
```

→ **Test : WpfEx1BisBinding**

Avec cette configuration le contrôle *TextBox* est lié aux données de l'élément de type *Slider* nommé *NbSlider*, propriété *Value* ; la liaison est bidirectionnelle. A l'exécution, on remarque que les modifications de la source se font seulement lorsque le contrôle *TextBox* perd le focus.

WPF : La classe Binding

Exemple TwoWay : (suite)

Pour que le contrôle *Slider* suive immédiatement la modification du contrôle *TextBox*, il faut modifier le Binding :

```
<Slider x:Name="NbSlider" HorizontalAlignment="Left" Margin="120,255,0,0"
VerticalAlignment="Top" Height="52" Width="275" Maximum="50" Value="25"/>

<TextBox x:Name="txtNb" HorizontalAlignment="Left" Height="23" Margin="550,255,0,0"
TextWrapping="Wrap" Text="{Binding ElementName=NbSlider,Path=Value,Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}" VerticalAlignment="Top" Width="120"/>
```

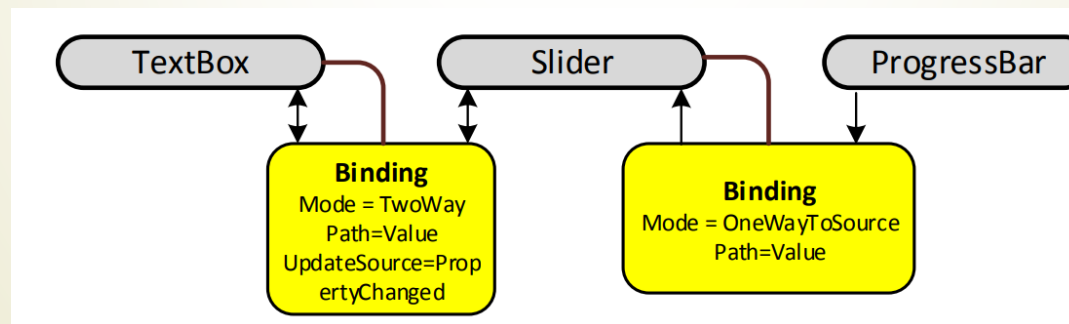
Le contrôle *TextBox* affiche un nombre réel si on déplace le contrôle *Slider* avec la souris. On peut modifier l'affichage pour obtenir une valeur entière comme ceci :

```
<TextBox x:Name="txtNb" HorizontalAlignment="Left" Height="23"
Margin="586,255,0,0" TextWrapping="Wrap" Text="{Binding ElementName=NbSlider,
Path=Value, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged,
StringFormat=\{0:N0\}}" VerticalAlignment="Top" Width="120"/>
```


WPF : La classe Binding

L'exemple suivant va permettre de chaîner un 3ème contrôle de type *ProgressBar* aux deux précédents.

Dans un premier temps nous allons associer le contrôle *Slider* à un contrôle *ProgressBar* (propriété Value) en mode *OneWayToSource*, le contrôle *ProgressBar* ne pourra pas être modifié par interaction avec l'utilisateur.



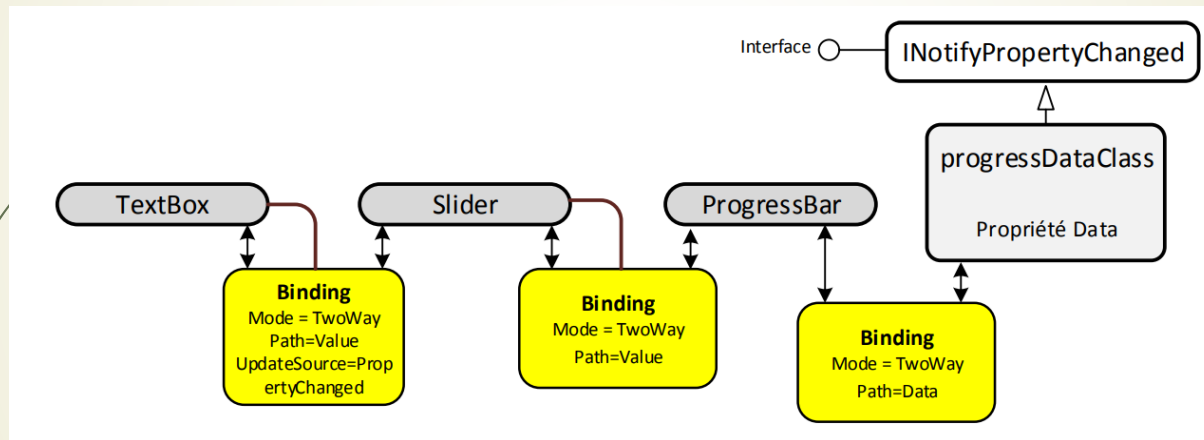
WPF : La classe Binding

```
<Grid>
  <Slider x:Name="NbSlider" HorizontalAlignment="Left" Margin="150,80,0,0"
    VerticalAlignment="Top" Height="50" Width="275" Maximum="50"
    TickPlacement="BottomRight" TickFrequency="5" Background="#FF9BDE91">
    <Slider.Value>
      <Binding ElementName="ProgressBarNb" Path="Value"
        Mode="OneWayToSource"/>
    </Slider.Value>
  </Slider>
  <TextBox x:Name="txtNb" HorizontalAlignment="Left" Height="50"
    Margin="590,80,0,0" TextWrapping="Wrap" Text="{Binding
    ElementName=NbSlider,Path=Value,Mode=TwoWay,
    UpdateSourceTrigger=PropertyChanged,StringFormat=\{0:N0\}}"
    VerticalAlignment="Top" Width="120" FontWeight="Bold"
    Background="#FF95D289"/>
  <ProgressBar x:Name="ProgressBarNb" HorizontalAlignment="Left" Height="50"
    Margin="290,210,0,0" VerticalAlignment="Top" Width="250"
    Maximum="50"/>
</Grid>
```

→ Test : *WpfEx2Binding*

WPF : Liaison avec un objet métier

Le but maintenant est de récupérer dans le code la valeur de la *Value* de la *ProgressBar*. Pour cela il faut réaliser une liaison de données entre une propriété d'un objet métier de l'application et la propriété *Value* du contrôle *ProgressBar*.



On voudrait également qu'une modification de la propriété de l'objet lié se répercute sur les contrôles de l'interface utilisateur. Pour cela il faut que la liaison soit bidirectionnelle partout c'est-à-dire que le binding entre le Slider et la ProgressBar doit être en mode TwoWay.

WPF : Liaison avec un objet métier

Interface *INotifyPropertyChanged* :

Pour lier la propriété *Value* de la *ProgressBar* de manière bidirectionnelle à un objet métier d'une classe, il faut que cette classe implémente l'interface *INotifyPropertyChanged*.

Cette interface permet de notifier à un client qu'une propriété a changé, elle reflète le changement à la source de données sans que la liaison soit réinitialisée.

C'est le cas en WPF quand on utilise du *Binding* sur un objet métier.

Quand on lie un composant graphique à un objet qui implémente cette interface, il enregistre un délégué sur l'événement *INotifyPropertyChanged.PropertyChanged*. Cet événement met à jour l'interface utilisateur en cas de changement de la propriété sans écrire de code supplémentaire.

WPF : Liaison avec un objet métier

```
class progressDataClass : INotifyPropertyChanged
{
    private double data;
    public event PropertyChangedEventHandler PropertyChanged;
    public double Data
    {
        get { return data; }
        set
        {
            if(data!=value)
            {
                data = value;
                NotifyPropertyChanged("Data");
            }
        }
    }
    public progressDataClass(double valeur)
    {
        this.data = valeur;
    }
    private void NotifyPropertyChanged(string v)
    {
        if (this.PropertyChanged != null)
            this.PropertyChanged(this, new PropertyChangedEventArgs(v));
    }
}
```

WPF : Liaison avec un objet métier

Une fois la classe métier créée, Il faut associer la propriété *Value* du contrôle *ProgressBar* à la propriété *Data* de cette classe métier.

Pour cela, il faut créer un objet ***ObjectDataProvider*** dans la section ressources de du fichier XAML :

```
<Window.Resources>
  <ObjectDataProvider x:Key="InstanceDonnée" ObjectType="{x:Type
local:progressDataClass }">
    <ObjectDataProvider.ConstructorParameters>
      <system:Double>30.0</system:Double>
    </ObjectDataProvider.ConstructorParameters>
  </ObjectDataProvider>
</Window.Resources>
```

WPF : Liaison avec un objet métier

```
<Slider x:Name="NbreSlider" HorizontalAlignment="Left" Margin="120,90,0,0"
    VerticalAlignment="Top" Height="35" Width="275" Maximum="50"
    TickPlacement="BottomRight" TickFrequency="5" Background="#FF6CAD6A">
    <Slider.Value>
        <Binding ElementName="ProgressBarNbre" Path="Value" Mode="TwoWay"/>
    </Slider.Value>
</Slider>
<TextBox x:Name="txtNbre" HorizontalAlignment="Left" Height="35" Margin="85,90,0,0"
    TextWrapping="Wrap" Text="{Binding ElementName=NbreSlider,
    Path=Value, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged, StringFormat=\{0:N0\}}"
    VerticalAlignment="Top" Width="115" RenderTransformOrigin="0.287,-3.728"
    FontWeight="Bold" Background="#FFA1DAD1"
    HorizontalContentAlignment="Center" VerticalContentAlignment="Center"
    Grid.Column="1"/>
<ProgressBar x:Name="ProgressBarNbre" HorizontalAlignment="Left" Height="35"
    Margin="136,38,0,0" VerticalAlignment="Top" Width="242" Maximum="50"
    Value="{Binding Source={StaticResource InstanceDonnée}, Mode=TwoWay, Path=Data}"
    Grid.Row="1" />
<Label x:Name="LblDonnee" Content="{Binding Source={StaticResource InstanceDonnée},
    Path=Data, Mode=OneWay}" Margin="85,40,130,0" Height="35" FontWeight="Bold"
    VerticalContentAlignment="Center" HorizontalContentAlignment="Center"
    Background="#FFA1DAD1" Grid.Row="1" Grid.Column="1" VerticalAlignment="Top" />
```

→ Test : *WpfEx4ObjetMetier*

WPF : Liaison avec un objet métier

L'exemple précédent a été réalisé entièrement en XAML, ce qui peut paraître fastidieux. On peut bien évidemment réaliser la même chose avec du code

```
public partial class MainWindow : Window
{
    private progressDataClass progressData = null;
    private Binding progressBinding = null;
    public MainWindow()
    {
        InitializeComponent();
        progressData = new progressDataClass(50.0);
        this.ProgressBarNbre.DataContext = progressData;
        progressBinding = new Binding("Data");
        progressBinding.Mode = BindingMode.TwoWay;
        ProgressBarNbre.SetBinding(ProgressBar.ValueProperty, progressBinding);
    }
    private void btnAfficher_Click_1(object sender, RoutedEventArgs e)
    {
        MessageBox.Show(this.txtNbre.Text);
        MessageBox.Show(string.Format("Valeur de la propriété Data : {0:0}",
            progressData.Data), "Valeur de l'attribut data");
        progressData.Data = 50.0;
    }
}
```

Avec code : → *Test : WpfEx5AvecCode*

WPF : Exécuter une méthode depuis XAML

Avec WPF, il est possible d'exécuter une méthode d'une classe métier sans ajout de code C#.

Dans l'exemple précédent (sans code), nous ajoutons une méthode *ModifierValeur* à la classe métier *progressDataClass* :

```
public void ModifierValeur(double val)
{
    if (val < 0)
        return;
    if (val > 50)
        return;
    Data = val;
}
```

La méthode sera déclarée dans la section *ressources* du fichier XAML et sera appelée par le contrôle *TextBox* :

```
<ObjectDataProvider ObjectInstance="{StaticResource InstanceDonnée}"
    x:Key="ModifierValeur" MethodName="ModifierValeur">
    <ObjectDataProvider.MethodParameters>
        <system:Double>0</system:Double>
    </ObjectDataProvider.MethodParameters>
</ObjectDataProvider>
```

WPF : Interface *IValueConverter*

La méthode *ModifierValeur* attend un paramètre de type double. Il va falloir convertir La valeur saisie dans le contrôle TextBox en double.

Pour cela il faut utiliser un convertisseur qui sera associé au "binding".

Ce convertisseur est réalisé par une classe qui implémente l'interface *IValueConverter*.

Cette interface implique de coder les méthodes *Convert* et *ConvertBack* qui permettent de passer d'un type à un autre.

```
class ConverterDoubleString : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        if (value != null)
            return string.Format("{0:N1}", value);
        return null; }
    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        string chaine = value as string;
        if (chaine != null) {
            double resultat;
            if (double.TryParse(chaine, out resultat))
                return resultat;
        }
        return null; } } }
```

WPF : Interface *IvalueConverter*

Pour utiliser cette classe il faut la déclarer dans la section *ressources* du fichier XAML de la manière suivante :

```
<local:ConverterDoubleString x:Key="DoubleString" />
```

Pour tester, on ajoute un contrôle *label* à la fenêtre et on récupère dedans la nouvelle valeur de l'objet

```
<Label    HorizontalAlignment="Left" Margin="10,30,0,0"  
    Grid.Row="3" VerticalAlignment="Top" Height="40"  
    Width="150" Background="#FFDEE8E7" Grid.Column="1">  
    <Label.Content>  
        <Binding Source="{StaticResource InstanceDonnée}"  
            Path="Data" >  
    </Binding>  
    </Label.Content>  
</Label>
```

→ Test : *WpfEx6Avecmethode*

WPF

WPF : Exécution d'une méthode

The screenshot shows a WPF application window titled "Exécution d'une méthode". The window contains a user interface for temperature conversion. It features a label "Entrez la température à convertir" followed by a text input field containing the value "0". To the right of the input field is a dropdown menu currently displaying "Celsius". Below these elements, the result is displayed in two teal-colored boxes: "Résultat :" and "32 Fahrenheit".

Exécution d'une méthode

Entrez la température à convertir

0

Celsius

Résultat :

32 Fahrenheit

WPF : Exécution d'une méthode

On définit une classe *TemperatureScale* :

```
internal class TemperatureScale : INotifyPropertyChanged
{
    private TempType _type;
    public TemperatureScale() { }
    public TemperatureScale(TempType type) { _type = type; }
    public TempType Type {
        get { return _type; }
        set
        {
            _type = value;
            OnPropertyChanged("Type");
        }
    }
    public event PropertyChangedEventHandler PropertyChanged;
    public string ConvertTemp(double degree, TempType temptype) {
        Type = temptype;
        switch (temptype)
        {
            case TempType.Celsius:
                return (degree * 9 / 5 + 32).ToString(CultureInfo.InvariantCulture) + " " + "Fahrenheit";
            case TempType.Fahrenheit:
                return ((degree - 32) / 9 * 5).ToString(CultureInfo.InvariantCulture) + " " + "Celsius";
        }
        return "Type inconnu";
    }
    protected void OnPropertyChanged(string name) {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
    }
}
```


WPF : Exécution d'une méthode

```
<Window.Resources>
  <ObjectDataProvider x:Key="TempConv" ObjectType="{ x:Type
    local:TemperatureScale}" MethodName="ConvertTemp" >
    <ObjectDataProvider.MethodParameters>
      <system:Double>0</system:Double>
      <local:TempType>Celsius</local:TempType>
    </ObjectDataProvider.MethodParameters>
  </ObjectDataProvider>
  <local:DoubleToString x:Key="DoubleToString" />
</Window.Resources>
```

```
<TextBox Grid.Row="1" Grid.Column="1" Name="tb" Margin="0,20,0,20"
Height="40" Background="#FFA4F5D7">
  <TextBox.Text>
    <Binding Source="{StaticResource TempConv}"
      Path="MethodParameters[0]"
      BindsDirectlyToSource="true" UpdateSourceTrigger="PropertyChanged"
      Converter="{StaticResource DoubleToString}">
    </Binding>
  </TextBox.Text>
</TextBox>
```

nom de la méthode

1^{er} paramètre

La méthode sera appelée dès qu'il y a une modification

Nom du convertisseur

WPF

WPF : Exécution d'une méthode

```
<ComboBox Grid.Row="1" Grid.Column="2"
    SelectedValue="{Binding Source={StaticResource TempConv},
        Path=MethodParameters[1], BindsDirectlyToSource=true}"
    Margin="10,0,0,0" Height="40" VerticalAlignment="Center"
    Background="#FF75B1B0" HorizontalAlignment="Left" Width="102">
    <local:TempType>Celsius</local:TempType>
    <local:TempType>Fahrenheit</local:TempType>
</ComboBox>
```

```
namespace WpfExTemperature
{
    public enum TempType
    {
        Celsius,
        Fahrenheit
    }
}
```

→ Test : WpfExTemperature

WPF : les règles de validation

La propriété *ValidationRules* permet de spécifier des objets de validation des données dans un élément.

Ces objets doivent être des instances de classes dérivées de la classe *ValidationRule*. Ces classes doivent surcharger la méthode abstraite *Validate*.

Syntaxe :

ValidationResult Validate(object value, System.Globalization.CultureInfo cultureInfo)

Value : objet à valider issu de la propriété sur laquelle on doit appliquer la règle.

cultureInfo : informations sur la culture à appliquer sur le contrôle, ce paramètre est utilisé si une propriété *ConverterCulture* de la classe *Binding* est spécifiée.

ValidationResult : type de retour qui est soit une instance définie par la propriété statique *ValidationResult.ValidResult* si la valeur est valide, soit une instance créée avec une propriété *IsValid* = *false* dans le cas contraire. Il est recommandé de préciser la raison de la non-validation dans la propriété *ErrorContent*.

WPF : les règles de validation

Dans l'exemple précédent on ajoute une règle de validation de la valeur de température saisie et on affiche une exception si la valeur est incorrecte. On ajoute alors la classe suivante :

```
class ValeurValidationRule : ValidationRule
{
    public override ValidationResult Validate(object value, CultureInfo cultureInfo)
    {
        double valeur = 0.00;

        try
        {
            if (((string)value).Length > 0)
                valeur = double.Parse((string)value);
        }
        catch (Exception e)
        {
            return new ValidationResult(false, "Caractères illégaux ou " + e.Message);
        }

        return new ValidationResult(true, null);
    }
}
```

WPF : les règles de validation

On modifie le TextBox pour tenir compte de la règle.

On peut également appliquer un style pour spécifier l'erreur

```
<TextBox Grid.Row="1" Name="tb" Margin="0,16,76,20" Background="#FFA4F5D7"
HorizontalAlignment="Right" Width="118" Grid.Column="1"<
```

Style

```
Validation.ErrorTemplate="{StaticResource validationTemp}"
Style="{StaticResource validationErreur}">
```

```
<TextBox.Text>
  <Binding Source="{StaticResource TempConv}" Path="MethodParameters[0]"
BindsDirectlyToSource="true" UpdateSourceTrigger="PropertyChanged"
Converter="{StaticResource DoubleToString}">
```

```
<Binding.ValidationRules>
  <local:ValeurValidationRule/>
</Binding.ValidationRules>
```

```
</Binding>
</TextBox.Text>
</TextBox>
```

Application de la règle

WPF : les règles de validation

```
<local:ValeurValidationRule x:Key="ValeurValidationRule"/>
```

```
<Style x:Key="validationErreur">  
  <Style.Triggers>  
    <Trigger Property="Validation.HasError" Value="True">  
      <Setter Property="Panel.Background" Value="Red"></Setter>  
      <Setter Property="FrameworkElement.ToolTip"  
        Value="{Binding RelativeSource={x:Static  
          RelativeSource.Self},Path=(Validation.Errors)[0].ErrorContent}" />  
    </Trigger>  
  </Style.Triggers>  
</Style>
```

```
<ControlTemplate x:Key="validationTemp">  
  <DockPanel>  
    <AdornedElementPlaceholder/>  
    <TextBlock Foreground="Red" FontWeight="Bold" FontSize="24">!!</TextBlock>  
  </DockPanel>  
</ControlTemplate>
```

→ Test : WpfEx7RegleValidation

WPF : Liaison aux données issues d'une base de données

Le principe reste le même : il faut coder une classe métier avec des méthodes réalisant l'accès aux données de la base de données.

Les données récupérées doivent être recopiées dans des propriétés, de la classe métier, accessibles par implémentation de l'interface *INotifyPropertyChanged*, avec autant d'événements *PropertyChangedEventHandler* qu'il y a de propriétés à lier dans la classe métier...

Visual Studio (depuis la version 2008 !) dispose d'un "Wizard" pour ajouter une classe du type LINQ to SQL, qui permet de créer une classe de liaison aux données d'une BDD SQL avec tout ce qu'il faut pour le "DataBinding" WPF.

WPF : Utilisation de Linq

LINQ *Language-INtegrated Query*

est un langage de requêtes simple qui fournit des fonctionnalités d'accès aux bases de données au niveau du langage et une API en C# et Visual Basic. C'est un composant du Framework.Net , sa syntaxe est proche du langage SQL. Linq intègre un langage de requête et offre un modèle cohérent pour travailler avec des données sur différents types de sources de données.



→ *Exemple d'utilisation: Accès à la base de données Rationnel*

WPF : Les styles et Triggers

Les triggers ([Trigger](#)) sont des déclencheurs. Leur rôle est de permettre de changer la valeur d'une ou plusieurs propriétés conditionnellement par déclenchement sur un trigger.

Il existe des triggers par propriétés, par données et par événements.

Ce mécanisme utilise largement l'objet [Style](#).

Selon les valeurs atteintes, le trigger se déclenche et la valeur d'autres propriétés change. On peut donc changer la valeur d'une propriété donnée, chaque fois qu'une certaine condition change.

Tous les contrôles WPF implémentent la notion de style. Les styles sont déclarés comme des ressources, ils sont définis soit directement dans la section ressources soit dans des fichiers de type *Dictionnaire de ressources*.

WPF : Les styles et Triggers

Exemple trigger par propriété :

```
<TextBlock Text="Bonjour à tous!" FontSize="32" HorizontalAlignment="Left"
VerticalAlignment="Top" Margin="124,123,0,0" Width="268" Height="39"
TextAlignment="Center" FontWeight="Bold" FontStyle="Italic">
  <TextBlock.Style>
    <Style TargetType="TextBlock">
      <Setter Property="Foreground" Value="Blue"></Setter>
      <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
          <Setter Property="Foreground" Value="Red" />
          <Setter Property="TextDecorations" Value="Underline" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </TextBlock.Style>
</TextBlock>
```

→ Test : *WpfExTriggerPropriete*

WPF : Les styles et Triggers

Exemple trigger par donnée :

créer un style personnalisé pour le type de contrôle " Label" : la couleur du contenu du label change selon la valeur d'une progressBar.

```
<Window.Resources>
  <Style x:Key="PassageEnGras" TargetType="{x:Type Label}">
    <Style.Triggers>
      <DataTrigger Binding="{Binding ElementName=Progressebarre,Path=Value}" Value="10" >
        <Setter Property="FontWeight" Value="Bold"></Setter>
        <Setter Property="Foreground" Value="Red"></Setter>
      </DataTrigger>
      <DataTrigger Binding="{Binding ElementName=Progressebarre,Path=Value}" Value="0" >
        <Setter Property="FontWeight" Value="SemiBold"></Setter>
        <Setter Property="Foreground" Value="Blue"></Setter>
      </DataTrigger>
      <DataTrigger Binding="{Binding ElementName=Progressebarre,Path=Value}" Value="5" >
        <Setter Property="FontWeight" Value="Bold"></Setter>
        <Setter Property="Foreground" Value="Green"></Setter>
      </DataTrigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

WPF : Les styles et Triggers

```
<ProgressBar x:Name="Progressebarre" HorizontalAlignment="Left" Height="42" Margin="74,52,0,0"
    VerticalAlignment="Top" Width="177" Foreground="#FF63BDA5" Maximum="10"
    RenderTransformOrigin="0.268,-1.036"/>
<Label Content="Valeur de la progressBar" Grid.Column="1" HorizontalAlignment="Left"
    Margin="39,52,0,0" VerticalAlignment="Top" Height="42" Width="146" Style="{StaticResource
    PassageEnGras}"/>
<Slider x:Name="SliderProg" HorizontalAlignment="Left" Margin="74,123,0,0"
    VerticalAlignment="Top" Height="33" Width="177" RenderTransformOrigin="0.215,0.515"
    Background="#FFADD68A">
    <Slider.Value>
        <Binding ElementName="Progressebarre" Path="Value" Mode="OneWayToSource"/>
    </Slider.Value>
</Slider>
<TextBox Grid.Column="2" HorizontalAlignment="Left" Height="25" Margin="10,52,0,0"
    TextWrapping="Wrap" Text="{Binding ElementName=Progressebarre, Path=Value }"
    VerticalAlignment="Top" Width="93"/>
```

→ Test : *WpfExTriggerData*

WPF : Les styles et Triggers

Exemple Trigger par évènement :

Les événements triggers, représentés par l'élément `<EventTrigger>`, sont principalement utilisés pour déclencher une animation, en réponse à l'appel d'un événement.

Dans l'exemple suivant, les événements de la souris *MouseEnter* et *MouseLeave* permettent respectivement de grossir la police du texte en 300 millisecondes et de la rapetisser plus lentement en 900 millisecondes.

```
<Grid>
  <TextBlock Name="lblStyled" Text="Bonjour à tous! C'est super d'utiliser les Triggers" FontSize="18"
HorizontalAlignment="Left" VerticalAlignment="Top" Margin="10,47,0,0" Width="571" FontWeight="Bold" FontStyle="Italic"
Height="125" TextWrapping="Wrap">
  <TextBlock.Style>
    <Style TargetType="TextBlock">
      <Style.Triggers>
        <EventTrigger RoutedEvent="MouseEnter">
          <EventTrigger.Actions>
            <BeginStoryboard>
              <Storyboard>
                <DoubleAnimation Duration="0:0:0.300" Storyboard.TargetProperty="FontSize" To="30" />
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger.Actions>
        </EventTrigger>
        <EventTrigger RoutedEvent="MouseLeave">
          <EventTrigger.Actions>
            <BeginStoryboard>
              <Storyboard>
                <DoubleAnimation Duration="0:0:0.900" Storyboard.TargetProperty="FontSize" To="15" />
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger.Actions>
        </EventTrigger>
      </Style.Triggers>
    </Style>
  </TextBlock.Style>
</TextBlock>
</Grid>
```

→ Test : *WpfExTriggerEvent*

WPF



WPF

<https://learn.microsoft.com/fr-fr/dotnet/desktop/wpf/controls>

FIN