



Technologie WPF / XAML

Cours basé sur les technologies WPF et XAML du Framework .NET autour du patron de conception MVVM

Présentation de WPF

- **WPF** : **W**indows **P**resentation **F**oundation.
- Permet la réalisation d'IHM (Interface Homme-Machine) tout comme son prédécesseur : Windows.Forms.
- Avantages des clients lourds :
 - Accès complet à l'ordinateur et au réseau.
 - Performance graphiques, accès à la 3D, flux vidéos, etc.
 - Interface graphique enrichie.
- Inconvénients des clients lourds :
 - Peu interopérable.
 - Installation nécessaire sur chaque poste (y compris pour les mises à jours).

Présentation de WPF

- WPF est une mutualisation de différentes API.



Types d'applications WPF

- On distingue deux types d'applications WPF :
 - Application « **stand alone** » ou « **smart application** » installée sur une ou plusieurs machines clientes.
 - Application « **navigateur** » téléchargée depuis internet et exécutée au travers d'un navigateur Web (Silverlight, XBAP).
- **ATTENTION** : Silverlight ne sera plus maintenu par Microsoft dans les années à venir!

Présentation de XAML

- **XAML** : e**X**tensible **A**pplication **M**arkup **L**anguage.
- Langage basé sur le XML.
- Permet de décrire les interfaces graphiques utilisées en WPF.
- Peut être réalisée par un designer via **Blend** par exemple qui génère ces fichiers XML.



Exemple de vue XAML

```

<Window x:Class="BankManagerUI.LoginWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Identification" Height="230" Width="350">

  <Grid Margin="15px">
    <Grid.RowDefinitions>
      <RowDefinition></RowDefinition>
      <RowDefinition></RowDefinition>
      <RowDefinition></RowDefinition>
      <RowDefinition></RowDefinition>
      <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition></ColumnDefinition>
      <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>

    <Label DockPanel.Dock="Top" Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2" FontSize="18" Foreground="Black" HorizontalAli
    <Label Margin="5px" Grid.Row="1" Grid.Column="0">Identifiant :</Label>
      <TextBox Margin="5px" Grid.Row="1" Grid.Column="1" Name="login"/>
      <Label Margin="5px" Grid.Row="2" Grid.Column="0">Mot de passe :</Label>
      <PasswordBox Margin="5px" Grid.Row="2" Grid.Column="1" Name="passwd"/>
      <CheckBox Grid.Row="3" Grid.ColumnSpan="2" Margin="5px" Name="firstConnexion">Je souhaite m'inscrire</CheckBox>
      <Button Grid.Row="4" Grid.Column="0" Margin="2px" Click="Click_Connect">Se connecter</Button>
      <Button Grid.Row="4" Grid.Column="1" Margin="2px" Click="Click_Quit">Quitter</Button>
    </Grid>
  </Window>

```

Philosophie de XAML

- XAML n'est qu'une **imbrication hiérarchique** de containers et de contrôles.
- L'objet **application** définit le point d'entrée d'un programme.
- Un programme est une succession de **Window**.
- Les principaux éléments de XAML sont :
 - Les éléments de positionnement (**containers**).
 - Les Widgets (**contrôles**).
 - Les éléments graphiques.

L'objet Application

- Importance de l'objet Application : un seul par application qui sert de « **démarrage** »
- **StartupUri** : Première fenêtre (Window) qui sera affichée à l'écran

```
<Application x:Class="Cours2_Containers.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="Stack.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

XAML

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

XAML.CS

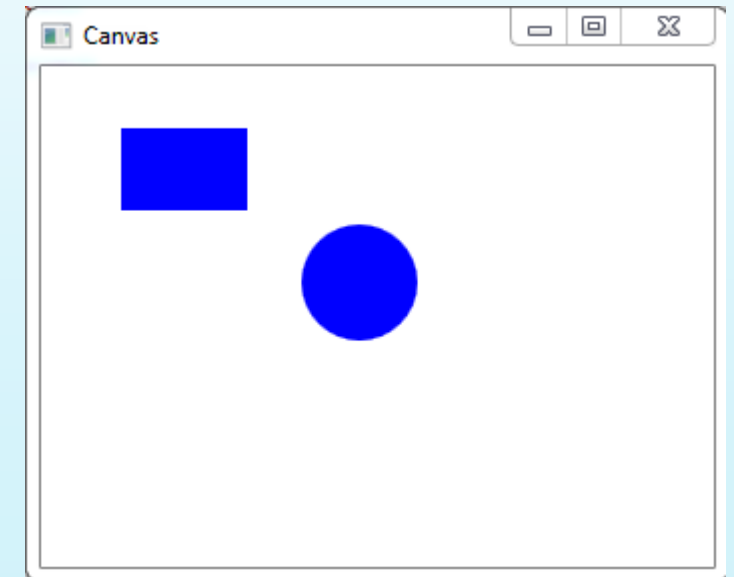
Les éléments de positionnement XAML

- Le conteneur de plus haut niveau dans une fenêtre est la **Window**. Un seul objet Window par fenêtre.
- Quelques exemples de **containers** intégrables dans une Window:
 - **Canvas** : Positionnement absolu.
 - **StackPanel** : Conteneur qui agence des éléments facilement.
 - **DockPanel** : Conteneur qui positionne selon les points cardinaux.
 - **GridPanel** : Conteneur qui se subdivise en lignes et colonnes.
- On peut imbriquer les container entre eux pour bénéficier d'un affichage personnalisé.

Canvas

- Très basique (voire trop basique...).
- Chaque contrôle doit définir Top / Left / Right / Bottom.
- Ne gère pas le redimensionnement de la fenêtre.

```
<Canvas>  
  <Rectangle Canvas.Left="40" Canvas.Top="31" Width="63" Height="41" Fill="Blue" />  
  <Ellipse Canvas.Left="130" Canvas.Top="79" Width="58" Height="58" Fill="Blue" />  
  <Path Canvas.Left="61" Canvas.Top="28" Width="133" Height="98" Fill="Blue"  
    Stretch="Fill" Data="M61,125 L193,28"/>  
</Canvas>
```



StackPanel

- Très utilisé, simple et pratique.
- Définition de l'orientation sur le StackPanel (vertical ou horizontal).

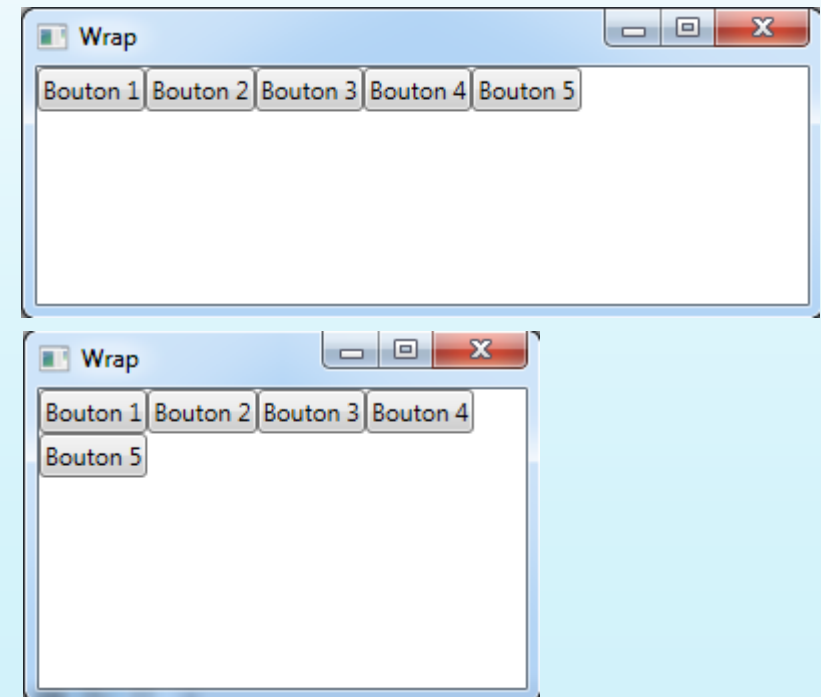
```
<StackPanel>  
  <TextBlock Margin="10" FontSize="20">Comment vous aimez votre café?</TextBlock>  
  <Button Margin="10">Noir sans sucre</Button>  
  <Button Margin="10">Avec un nuage de lait</Button>  
  <Button Margin="10">Capuccino</Button>  
  <Button Margin="10">Décaféiné</Button>  
</StackPanel>
```

The UI rendering shows a light blue rectangular container. At the top is a text prompt "Comment vous aimez votre café?" in a light blue box. Below it are four buttons stacked vertically, each with a light blue border and a light gray gradient background. The buttons are labeled "Noir sans sucre", "Avec un nuage de lait", "Capuccino", and "Décaféiné" from top to bottom.

WrapPanel

- Similaire au StackPanel.
- Permet de réarranger les contrôles en fonction de la place disponible dans le container parent.

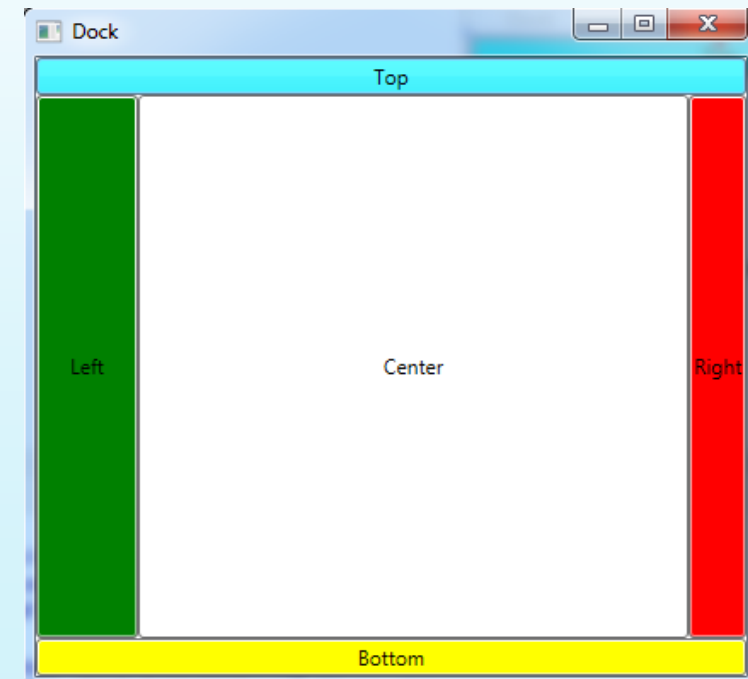
```
<WrapPanel Orientation="Horizontal">  
  <Button Content="Bouton 1" />  
  <Button Content="Bouton 2" />  
  <Button Content="Bouton 3" />  
  <Button Content="Bouton 4" />  
  <Button Content="Bouton 5" />  
</WrapPanel>
```



DockPanel

- Plus évolué que le StackPanel.
- Permet l'arrangement de cinq directions (Bottom, Top, Left, Right, Center).

```
<DockPanel LastChildFill="True">  
  <Button Content="Top" Background="Aqua" DockPanel.Dock="Top"/>  
  <Button Content="Bottom" Background="Yellow" DockPanel.Dock="Bottom"/>  
  <Button Content="Left" Width="60" Background="Green" DockPanel.Dock="Left"/>  
  <Button Content="Right" Background="Red" DockPanel.Dock="Right"/>  
  <Button Content="Center" Background="White"/>  
</DockPanel>
```



GridPanel

- Conteneur le plus customisable proposé par WPF.
- Permet de définir des colonnes et des lignes pour diviser l'écran.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
    <RowDefinition Height="28" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="200" />
  </Grid.ColumnDefinitions>
  <Label Grid.Row="0" Grid.Column="0" Content="Nom:" />
  <Label Grid.Row="1" Grid.Column="0" Content="Mail:" />
  <Label Grid.Row="2" Grid.Column="0" Content="Commentaire:" />
  <TextBox Grid.Column="1" Grid.Row="0" Margin="3" />
  <TextBox Grid.Column="1" Grid.Row="1" Margin="3" />
  <TextBox Grid.Column="1" Grid.Row="2" Margin="3" />
  <Button Grid.Column="1" Grid.Row="3" HorizontalAlignment="Right"
    MinWidth="80" Margin="3" Content="Envoyer" />
</Grid>
```



Les Widgets XAML

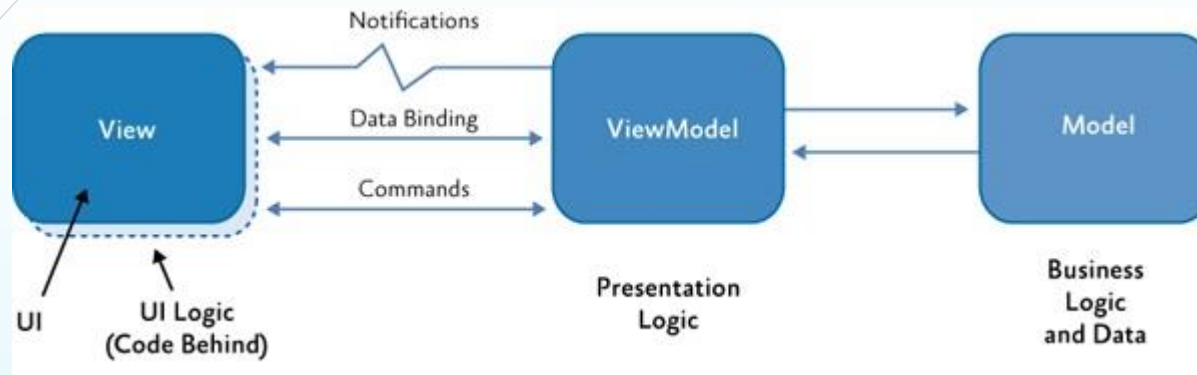
- Quelques exemples de **Widgets** :
 - **Button** : Bouton
 - **CheckBox** : Case à cocher.
 - **ComboBox** : contenant des ComboBoxItem.
 - **Image**.
 - **RadioButton** : bouton radio.
 - **ScrollBar** : barre de défilement.
 - **TextBox** : Champ de texte à taper.
 - **VerticalSlider** : ascenseur.
 - **Window** : fenêtre.
 - **ListBox** : liste de choix.
- Un Widget et un contrôle que l'on place dans un container.

Présentation de MVVM

MVVM est un design pattern adapté pour le développement d'applications basées sur les technologies WPF. Il est intéressant à mettre en place dans les projets pour plusieurs raisons :

- Le **faible couplage** entre la Vue et le Vue-Modèle permet de pouvoir modifier facilement la vue sans avoir d'impact sur le Vue-Modèle (et vice versa).
- Il permet de tester de manière séparée les différents éléments de la solution.
- Il permet une maintenance facilitée des projets.
- Il permet une meilleure **réutilisation** de code.

Présentation de MVVM



- Schéma tiré du MSDN.
- Le bloc Vue-Modèle peut être vu comme un **adaptateur** entre la Vue et Modèle.

MVVM : Le Modèle

- Constitue la couche d'accès aux données.
- Est réalisé en C#.
- Ne connaît pas l'interface graphique qui lui est associé.

Le modèle correspond à notre couche développée avec **Entity Framework** dans le cours précédent.

On pourra accéder aux objets du modèle et à la base de données à travers la **BLL**.

MVVM : La vue

- Est constituée des éléments graphiques (containers, contrôles, etc.).
- Prends en charge l'interaction avec l'utilisateur (clavier, souris, webcam par exemple).
- Notion de **DataBinding** permet de présenter des données dans une vue.
- Pas de code dans le **CodeBehind**.
- Ne dois **jamais** traiter de données (juste de l'affichage).

MVVM : Le Vue-Modèle

- Est un **adaptateur** entre la vue et le modèle.
- Est responsable des tâches suivantes :
 - Adapter des types du modèle en des types exploitables via le **DataBinding** par la vue.
 - Exposer des **commandes** que la vue pourra utiliser pour interagir avec le modèle.
- Peut être testée par des tests unitaires.
- N'a pas nécessairement besoin de connaître la vue qui les exploite.

L'interface INotifyPropertyChanged

- **Interface** fournie dans le Framework.NET (System.ComponentModel).
- Informe le Vue-Modèle de tous les **changements** effectués sur les propriétés de la vue.
- Tous les Vues-Modèles doivent implémenter cette interface.

```
namespace System.ComponentModel
{
    // Summary:
    //     Notifies clients that a property value has changed.
    public interface INotifyPropertyChanged
    {
        // Summary:
        //     Occurs when a property value changes.
        event PropertyChangedEventHandler PropertyChanged;
    }
}
```

Une classe de base pour les Vues-Modèles

- Implémente l'interface **INotifyPropertyChanged**.
- Est abstraite.
- Tous les Vues-Modèles sont conçus de la même façon (**héritage**).
- On évite la duplication de code.

```
/// <summary>
/// Classe de base pour tous les Vues-Modèles de l'application.
/// Elle fournit la gestion du PropertyChanged via l'interface INotifyPropertyChanged.
/// Cette classe est abstraite.
/// </summary>
public abstract class ViewModelBase : INotifyPropertyChanged
{
    #region Constructeurs

    protected ViewModelBase()
    {
    }

    #endregion

    #region Membres de INotifyPropertyChanged

    /// <summary>
    /// Se déclenche lorsque une propriété de cet objet a une nouvelle valeur.
    /// </summary>
    public event PropertyChangedEventHandler PropertyChanged;

    /// <summary>
    /// Déclenche l'évènement sur l'évènement PropertyChanged event.
    /// </summary>
    /// <param name="propertyName">La propriété qui a une nouvelle valeur</param>
    protected virtual void OnPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = this.PropertyChanged;
        if (handler != null)
        {
            var e = new PropertyChangedEventArgs(propertyName);
            handler(this, e);
        }
    }

    #endregion
}
```

Instanciation des classes Vue-Modèle

- Lorsqu'une Vue est créée, il est nécessaire de créer la classe Vue-Modèle qui lui est associée. Il est possible de réaliser cela de deux manières :
 - Depuis le code-behind en C#, on instancie la classe Vue-Modèle dans le constructeur et on définit le **DataContext** de la vue.

```
/// <summary>
/// Constructeur de la Fenêtre principale
/// </summary>
public MainWindow()
{
    InitializeComponent();
    this.DataContext = new HomeViewModel();
}
```

- Directement depuis le XAML de la Vue, on définit la propriété **DataContext**.

```
<UserControl x:Class="Cours2_ExempleMVVM.Views.ListeProduit"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:my="clr-namespace:Cours2_ExempleMVVM.Views"
    xmlns:local="clr-namespace:Cours2_ExempleMVVM.ViewModels"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="525">
    <UserControl.DataContext>
        <local:ListeProduitViewModel></local:ListeProduitViewModel>
    </UserControl.DataContext>
    <Grid...>
</UserControl>
```

DataBinding

- Notion **ultra-importante** dans le mécanisme MVVM.
- Permet de lier une propriété du VM lié à la vue à une propriété d'un élément.
- Exemple :

```
<StackPanel Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="2" Orientation="Vertical">
    <Label>Code du produit</Label>
    <TextBox Text="{Binding Code}"></TextBox>
    <Label>Nom du produit</Label>
    <TextBox Text="{Binding Nom}"></TextBox>
</StackPanel>
```



```
/// <summary>
/// ViewModel représentant un Produit (avec son détail)
/// Hérite de BaseViewModel
/// </summary>
public class DetailProduitViewModel : BaseViewModel
{
    Variables

    Constructeurs

    #region Data Bindings

    /// <summary>
    /// Code du produit
    /// </summary>
    public string Code
    {
        get { return _code; }
        set { _code = value; }
    }

    /// <summary>
    /// Nom du produit
    /// </summary>
    public string Nom
    {
        get { return _nom; }
        set { _nom = value; }
    }

    #endregion

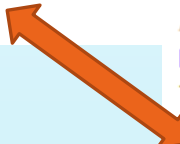
    Commandes
}
```


DataBinding

- Dans le cas d'un binding vers une liste d'objets, il faut implémenter l'interface **INotifyCollectionChanged**.
- WPF fournit sa propre implémentation : la classe **ObservableCollection** pour éviter d'implémenter cette interface.
- Exemple :

```
<ListView ItemsSource="{Binding Produits, UpdateSourceTrigger=PropertyChanged}">
  <ListView.View>
    <GridView>
      <GridViewColumn Width="100px" Header="Code" DisplayMemberBinding="{Binding Code}" />
      <GridViewColumn Width="150px" Header="Nom" DisplayMemberBinding="{Binding Nom}" />
    </GridView>
  </ListView.View>
</ListView>
```

```
/// <summary>
/// Obtient ou définit une collection de DetailProduitViewModel (Observable)
/// </summary>
public ObservableCollection<DetailProduitViewModel> Produits
{
    get { return _produits; }
    set
    {
        _produits = value;
        OnPropertyChanged("Produits");
    }
}
```



DataBinding

- La propriété **Binding.UpdateSourceTrigger** permet de déterminer le timing des mises à jour de la source de liaison
 - **PropertyChanged** quand la valeur de la propriété change.
 - **LostFocus** quand l'élément perd le focus (exemple : TextBox).
- Le **BindingMode** décrit le sens de données dans une liaison :
 - **OneWay** : Met à jour la propriété cible de liaison (cible) lorsque la source de liaison (source) est modifiée
 - **TwoWay** : Entraîne la mise à jour automatique de la propriété source lorsque la propriété cible est modifiée et inversement
 - **OneTime** : Met à jour la cible de liaison lorsque l'application démarre ou lorsque le contexte de données est modifié
 - **Default** : Par défaut, c'est le plus souvent le mode TwoWay (mais cela peut varier!)

DataBinding

- On peut aussi binder le **DataContext** pour associer un ViewModel à une vue.

```
<my:DetailProduit DataContext="{Binding SelectedProduit, UpdateSourceTrigger=PropertyChanged}"></my:DetailProduit>
```

- Exemple :
 - Ecran maitre-détail pour afficher l'élément sélectionné en live (voir projet d'exemple).

Les commandes

- Avec MVVM, il est **interdit** de lier un bouton à une méthode ou de gérer son évènement dans le code Behind de la vue. Solution : l'interface **ICommand**.
- Définir un objet pour gérer toutes les commandes qui va hériter de cette interface : le **RelayCommand** (voir projet de test).
- La classe **RelayCommand** sert de « lanceur » de commande et ce quelques soit leur type (délégué de type Action).

IMPORTANT :

Il faut définir un élément de type **ICommand** qui sera lié au bouton et enfin **définir le comportement** de cette commande via l'objet **RelayCommand**.

Les commandes : Exemple

- Élément de type ICommand dans le Vue-Modèle.

```
private RelaiComande _actionSave;  
public ICommand ActionSave  
{  
    get  
    {  
        if (_actionSave == null)  
            _actionSave = new RelaiComande(() => this.AddOperation());  
        return _actionSave;  
    }  
}
```

- Définition de mon comportement

```
private void AddOperation()  
{  
    // comportement à définir  
}
```





- Lier la commande au bouton

```
<Button Margin="5" Height="25px" Grid.Row="0" Grid.Column="0" Command="{Binding ActionSave}"
```

- Exemple de comportement : Ouvrir une popup, ajouter une donnée en base, modifier une donnée en base, afficher une erreur, etc..

User Control

- Egalement appelé « contrôle utilisateur ».
- Fournit un moyen très simple de créer un **contrôle réutilisable** dans des Window ou dans d'autres UserControl.
- **ATTENTION** : Ne pas confondre avec le UserControl d'ASP WebForm (même nom mais pas le même assembly).

	Fenêtre (WPF)	Visual C#
	Page (WPF)	Visual C#
	Contrôle utilisateur (WPF)	Visual C#
	Dictionnaire de ressources (WPF)	Visual C#

UserControl

- **Privilégier** l'utilisation des contrôles utilisateurs pour réutiliser vos vues. Il suffit ensuite de changer la liaison avec le Vue Modèle.
- Propriété **DataContext** : Permet de lier un ViewModel pour le DataBinding.
- Peut être placée dans une Window pour en faire un popup.

```
Views.Operation operationWindow = new Views.Operation();  
operationWindow.DataContext = this;  
operationWindow.ShowDialog();
```

```
Window window = new Window  
{  
    Title = "My User Control Dialog",  
    Content = new MyUserControl()  
};
```

- **ATTENTION** : ShowDialog s'applique sur une Window!

Namespace

- Pour utiliser des contrôles personnalisés et tiers en XAML, il faut créer un **espace de nom** (ou namespace).
- On parle « **d'importer** » un espace de nom.
- Utile par exemple pour intégrer un **UserControl** dans une Window.
- Utile par exemple aussi pour lier un VM dans le XAML.

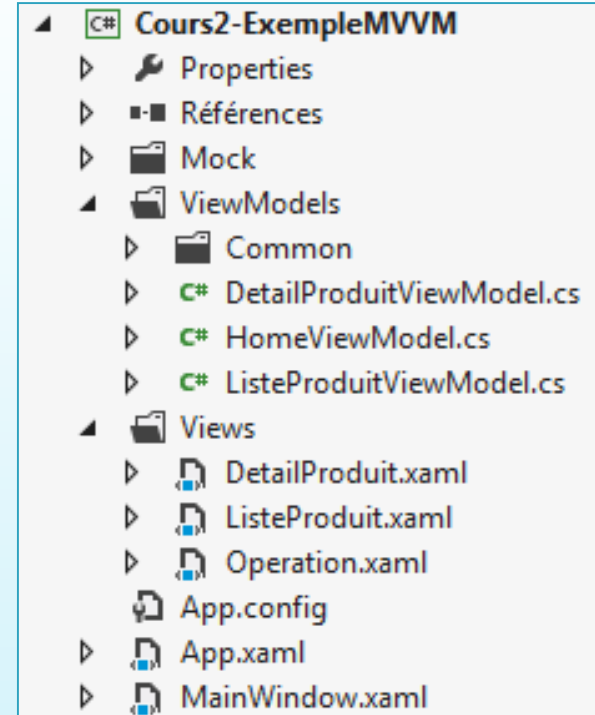
```
xmlns:my="clr-namespace:Cours2_ExempleMVVM.Views"
```

Nom de votre
namespace

Namespace à importer

Architecture d'un projet

- Un répertoire **ViewModels** pour placer les vues modèles.
- Un répertoire Views pour placer les vues.
- Une App.xaml (point d'entrée de l'application).
- Une MainWindow.xaml (Window principale de l'application).
- **NB** : Dans les références, il faudra ajouter les DLL pour référencer votre modèle et votre Business Layer.



Conclusion : MVVM – WPF - XAML

➤ Vue (XAML)

- Que du **XAML**
- **DataContext** défini sur l'objet VM
- **DataBinding** vers la classe VM (commandes & données)



➤ Vue Modèle (C# / WPF)

- **Adapteur** entre la vue et le modèle
- **Englobe** le modèle
- Implémente **INotifyPropertyChanged**
- Expose des **commandes**
- Est en charge de la **validation**
- **Testable**



➤ Modèle (C# / EF)

- Pas de concept WPF
- Gère la **persistance** des données
- Couche **indépendante** qui peut être développée à part et réutilisable
- **Testable**
- **BLL / EF** dans notre cas

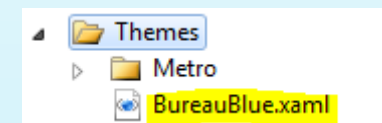
Les thèmes

- WPF inclut tous les thèmes Windows « classiques ».
- On peut **ajouter** son propre thème.
- De nombreux thèmes existent sur le web mais vous pouvez aussi en créer un (avec beaucoup de patience).

App.xaml :

```
<Application.Resources>  
  <ResourceDictionary Source="Themes/BureauBlue.xaml"/>  
</Application.Resources>
```

Répertoire Themes dans votre solution



- **NB** : Pensez aux thèmes disponibles dans les Nugets

Le menu

- Vous pouvez ajouter un **menu** à votre application
- Un menu peut être placée dans une Window, un UserControl, etc.
- Contrôle WPF : **TabControl** qui contient une liste de **TabItem**.

```
<TabControl Grid.Row="0" Grid.ColumnSpan="3">
  <TabItem Header="MON PROFIL">
    <TabItem.Content>
      <my:Profil DataContext="{Binding ProfilViewModel}" Margin="2" Grid.Row="0"></my:Profil>
    </TabItem.Content>
  </TabItem>
  <TabItem Header="MES COMPTES">
    <TabItem.Content>
      <my:Compte DataContext="{Binding CompteViewModel}" Margin="2" Grid.Row="0"></my:Compte>
    </TabItem.Content>
  </TabItem>
  <TabItem Header="MES CATEGORIES">
    <TabItem.Content>
      <my:Categorie DataContext="{Binding CategorieViewModel}" Margin="2" Grid.Row="0"></my:Categorie>
    </TabItem.Content>
  </TabItem>
</TabControl>
```

Pour aller plus loin

- Il existe de nombreux Framework qui implémentent déjà le patron MVVM et donc qui facilite sa mise en place
 - MvvmLight
 - Prism
 - MvvmCross

Travaux dirigé autour du TP

- Réalisation d'une interface avec Maître – Détail
 - Définir les vues à créer.
 - Définir les ViewModels à créer.
 - Relation avec la couche Modèle.
 - Découpage.
 - ...
- Objectif : Afficher une liste de produit à droite, un détail à gauche

Solution : Les vues

- Une vue MainWindow.xaml
 - Conteneur d'une vue contenant un contrôle utilisateur avec notre relation maître détail

```
<Window x:Class="Cours2_ExempleMVVM.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:my="clr-namespace:Cours2_ExempleMVVM.Views"
        Title="MainWindow" Height="450" Width="825">
    <Grid>
        <!-- Définition du gridLayout -->
        <Grid.RowDefinitions>
            <RowDefinition Height="50px"></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <DockPanel Background="Blue">
            <Label VerticalAlignment="Center" HorizontalAlignment="Center" FontWeight="Bold" FontSize="26"
                Content="Maître" />
        </DockPanel>
        <my:ListeProduit Grid.Row="1" DataContext="{Binding ListeProduitViewModel}" /></my:ListeProduit>
    </Grid>
</Window>
```

Solution : Les vues

- Un contrôle utilisateur ListeProduit.xaml
 - Contient la liste des produits avec le DataBinding nécessaire
 - Contient le détail du produit sélectionné avec le DataBinding nécessaire



Solution : Les vues

- Un contrôle utilisateur DetailProduit.xaml
 - Contient le détail d'un produit (toutes les propriétés sous forme de contrôle style dropdown, textbox, etc)
 - Utilisation d'une vue de disposition adéquate
 - Réutilisable
 - Gestion du DataBinding

Solution : Les ViewModels

- Un ViewModel de base
 - Encapsuler la gestion du **INotifyPropertyChanged**
- Un ViewModel **ListeProduitViewModel**
 - Hérite du ViewModel de base
 - Gère une collection de **DetailProduitViewModel**
 - Gère le **DetailProduitViewModel** sélectionné
- Un ViewModel **DetailProduitViewModel**
 - Gère le binding de toutes les propriétés d'un produit

Solution : Les ViewModels

