```
<Button Content="Prof-MTI">
    <Button.Triggers>
        <EventTrigger RoutedEvent="Button.Click">
            <EventTrigger.Actions>
                <BeginStoryboard Storyboard="{StaticResource StudentLearning}"/>
```

COURS WPF
PART 4

Lemettre Arnaud

Arnaud.lemettre@gmail.com

ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

Microsoft® .NET

Sommaire

- DataGrid
- JumpList
- InputBinding
- Autre

HIERARCHICAL DATATEMPLATE

HIERARCHICAL DATATEMPLATE

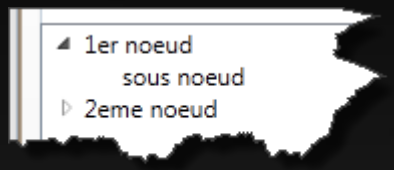
- Permet de faire des templates pour les objets prenant en charge HeaderedItemsControl

  ex :

  - Treeview
  - MenuItem

Hierarchical DataTemplate

- Le composant TreeView




```
<TreeView Grid.Row="1" x:Name="tree">
    <TreeViewItem Header="1er noeud" IsExpanded="True">
        <TreeViewItem Header="sous noeud"  />
    </TreeViewItem>
    <TreeViewItem Header="2eme noeud">
        <TreeViewItem Header="sous noeud" />
    </TreeViewItem>
</TreeView>
```

On peut manipuler les éléments du treeview par le code, mais l'ajout de sous éléments devient plus compliqué.

Cependant grâce au binding on peut manipuler les données du treeview.

Hierarchical DataTemplate

- La 1ère étape est de créer la classe permettant de représenter les valeurs de notre arbre.

```
public class Branches
{
    public string Name { get; set; }
    public ObservableCollection<Branches> SubBranch { get; set; }

    public Branches()
    {
        SubBranch = new ObservableCollection<Branches>();
    }

    public Branches(string name, ICollection<Branches> col)
    {
        Name = name;
        SubBranch = new ObservableCollection<Branches>(col);
    }
}
```

L'utilisation d'observableCollection

permettra de notifier les interfaces

dès l'ajout / suppression d'un noeud

HIERARCHICAL DATATEMPLATE

- La 2ème étape est de créer le template pour l'adapter à notre objet

```
<Window.Resources>
    <HierarchicalDataTemplate x:Key="brancheTemplate"
        ItemsSource="{Binding SubBranch}" >
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding Name}" />
        </StackPanel>
    </HierarchicalDataTemplate>
</Window.Resources>
```

Liste des sous éléments

Nom à afficher

Utilisation du template :

```
<TreeView Grid.Row="1" x:Name="tree"     ItemTemplate="{StaticResource brancheTemplate}"/>
```

Initialisation dans un contexte hors mvvm

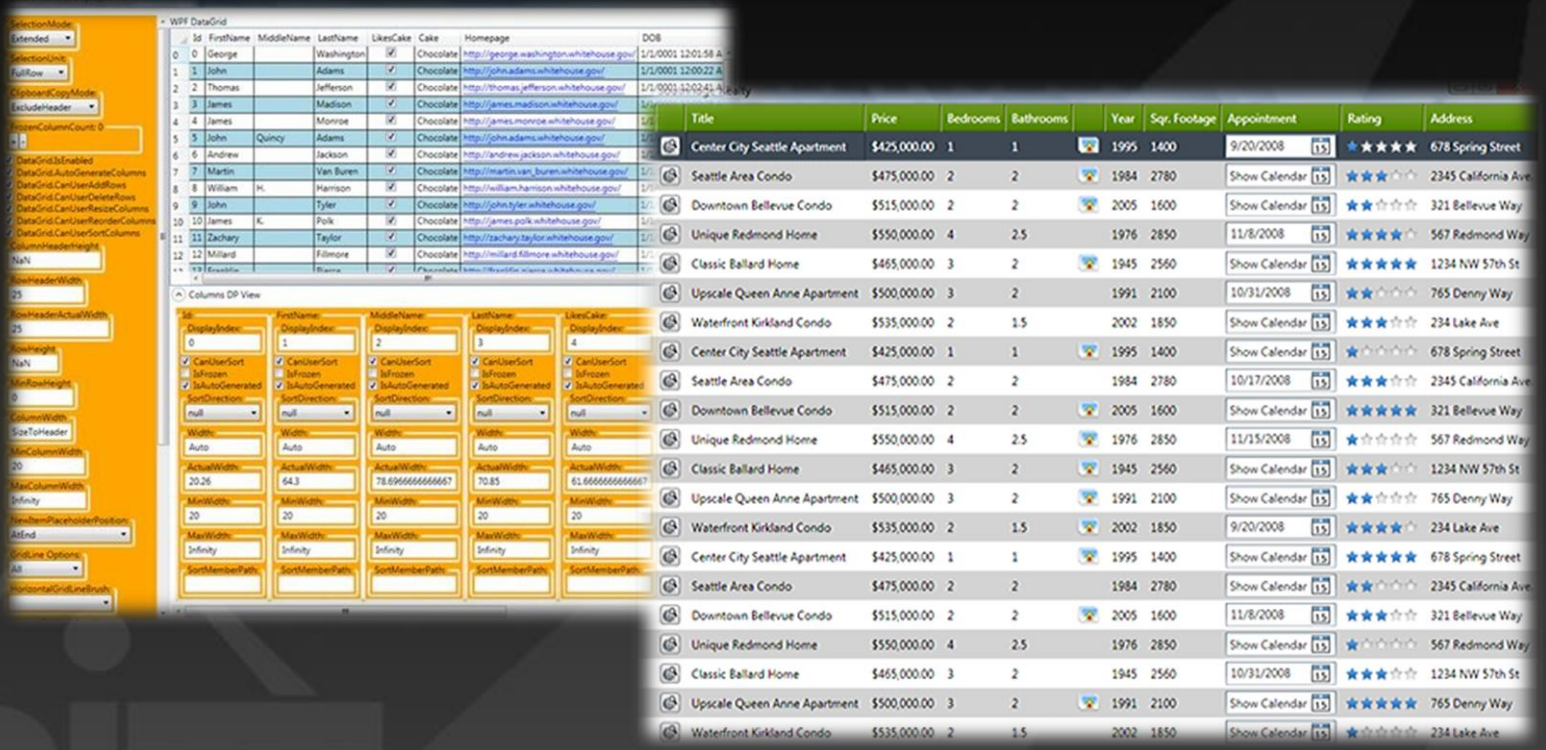
```
private ObservableCollection<Branches> nodes = new ObservableCollection<Branches>();

private void InitData()
{
    //1er noeud
    Branches firstNode = new Branches() { Name = "FirstNode" };
    firstNode.SubBranch.Add(new Branches() { Name = "SubNode" });
    //liste pour un sous noeud
    List<Branches> list = new List<Branches>();
    list.Add(new Branches() { Name = "First SubNode" });
    //second noeud
    Branches secondNode = new Branches("SecondNode", list);
    secondNode.SubBranch.Add(new Branches() { Name = "SubNode" });
    //ajout des noeuds
    nodes.Add(firstNode);
    nodes.Add(secondNode);
    //on affecte sur le composant
    tree.ItemsSource = nodes;
}
```

DATAGRID

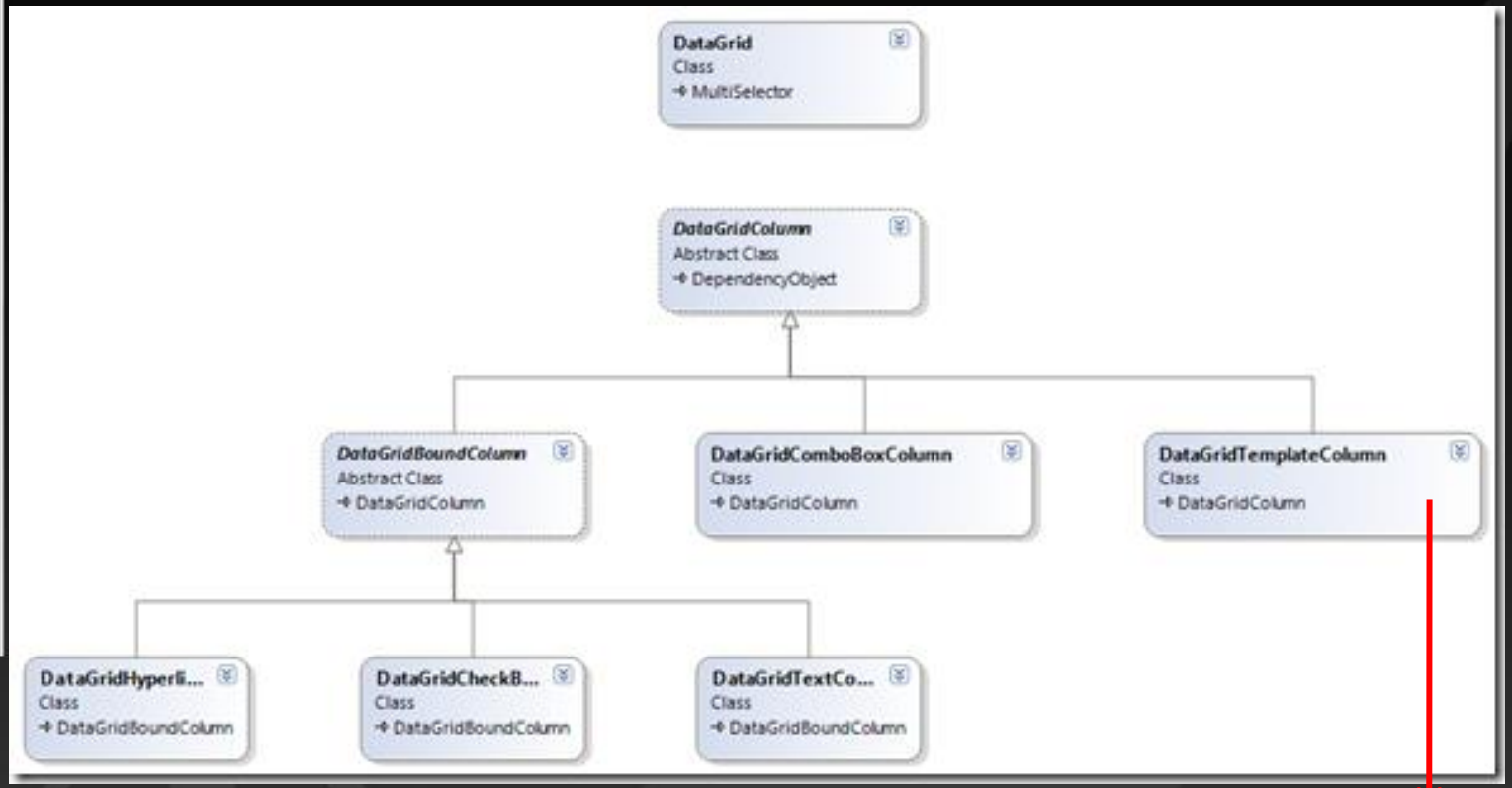
DATAGRID

- Pourquoi utiliser une datagrid ?
  - Cela permet de présenter un ensemble de données de manière tabulaire
  - De pouvoir laisser manipuler les données par les utilisateurs.



DATAGRID

- Un peu de technique et de classes utiles :

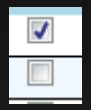


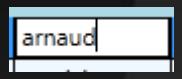
Permet de créer de nouveaux types de colonnes, pour afficher par exemple des images

DATAGRID

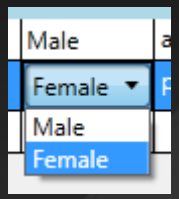
- Les différents types de colonnes :

DataGridBoundColumn.
DataGridHyperLinkColumn, DataGridCheckBoxColumn, DataGridTextColumn





DataGridComboBoxColumn



DATAGRID

Et en pratique …

Le code XAML => pour déclarer une datagrid :

```
<DataGrid ItemsSource="{Binding}" Name="dataGrid1" >

</DataGrid>
```

Il nous faut maintenant alimenter notre dataGrid :

```
public List<User> List {get; set ;}

public MainWindow()
{
    InitializeComponent();

    InitData();
    dataGrid1.DataContext = List;
}
```

Résultat :

Present	Gender	Firstname	Mail	Name
☑	Male	arnaud	mailto:arnaud.lemettre@gmail.com	lemettre
☐	Female	patricia	mailto:pat.cornwail@aol.com	cornwell
☐				

DATAGRID

Notre datagrid nous a affiché les données de manière automatique, néanmoins on peut choisir soit un affichage partiel des données ou certains types de colonne.

La propriété à impacter est AutoGenerateColumns :
-False : il faut mettre les colonnes manuellement
-True : le contrôle inspecte les données pour déterminer le nom / type de colonne.

```
<DataGrid ItemsSource="{Binding}" AutoGenerateColumns="False" Name="dataGrid1">
    <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding Present}" Header="Present" />
    </DataGrid.Columns>
</DataGrid>
```

Cependant les 2 modes peuvent fonctionner ensemble.

DATAGRID

Comme tout composant en WPF, on peut personnaliser le rendu graphique.

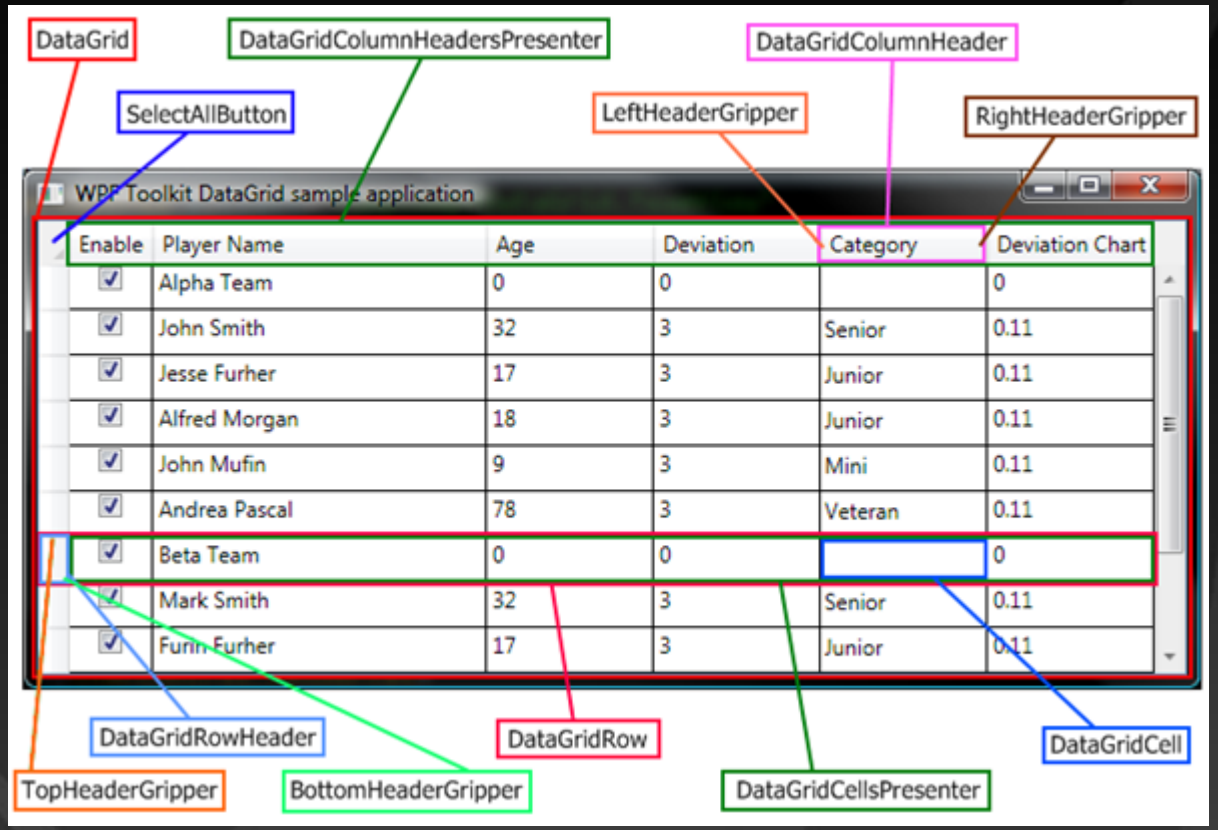
Via les styles :

On peut customiser les entêtes, ainsi que la cellule, ou la manière d'afficher des informations.

```
<!-- DataGrid style -->
<Style x:Key="DataGridStyle1" TargetType="{x:Type DataGrid}">
    <Setter Property="ColumnHeaderStyle" Value="{DynamicResource ColumnHe
aderStyle1}"/>
</Style>
<!-- DataGridColumnHeader style -->
<Style x:Key="ColumnHeaderStyle1" TargetType="DataGridColumnHeader">
    <Setter Property="Height" Value="30"/>
    <Setter Property="Background" Value="LightBlue"/>
    <Setter Property="Foreground" Value="Blue"/>
    <Setter Property="FontSize" Value="18" />
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="ToolTip" Value="Click to sort."/>
        </Trigger>
    </Style.Triggers>
</Style>
```

DataGrid

- En ce qui concerne le style de la dataGrid, les différentes propriétés sont représentées sur l'image suivante :



Article sur le style des datagrids

DATAGRID

Un peu plus de détails …

On peut adapter notre grid view dans le but d'avoir un mode détails sur les lignes

```
<DataGrid ItemsSource="{Binding}" AutoGenerateColumns="True" >
    <DataGrid.RowDetailsTemplate>
        <DataTemplate>
            <TextBlock Height="100" Text="{Binding Name}" />
        </DataTemplate>
    </DataGrid.RowDetailsTemplate>
</DataGrid>
```

DATAGRID

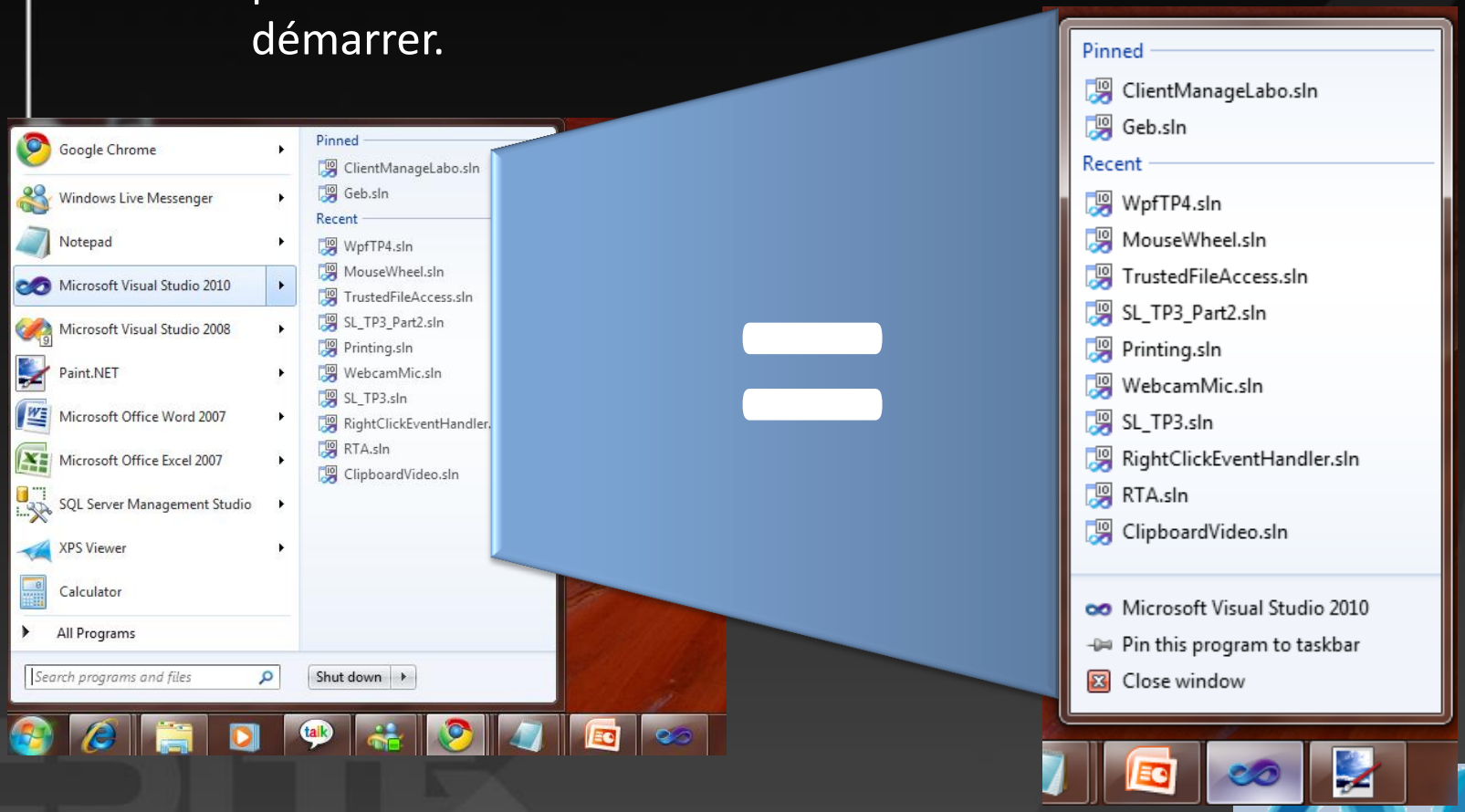
- Au travers de ce composant l'utilisateur peut également interagir avec les données. A cet effet plusieurs events existent:
  - CanUserAddRows, CanUserDeleteRows, CanUserReorderColumns, CanUserResizeColumns, CanUserResizeRows et CanUserSortColumns

Référence sur la msdn

JumpList

JumpList

- Présentation :
  - Apparue avec windows 7 les jumplist sont des raccourcis que l'on peut mettre sur la barre des tâches mais aussi dans le menu démarrer.



JumpList

- Mais que peut on faire avec ces nouvelles interactions ?
  - Communiquer l'état de l'application
  - Envoyer des informations à l'application
  - Rajouter des liens
  - …

JumpList

Au travers du XAML, on peut le mettre dans l'app.xaml :

```
<Application x:Class="WpfJumpList.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <JumpList.JumpList>
        <JumpList ShowRecentCategory="True"
                  ShowFrequentCategory="True"
                  JumpItemsRejected="JumpList_JumpItemsRejected"
                  JumpItemsRemovedByUser="JumpList_JumpItemsRemovedByUser">
            <JumpTask Title="Notepad"
                      Description="Open Notepad."
                      ApplicationPath="C:\Windows\notepad.exe"
                      IconResourcePath="C:\Windows\notepad.exe"/>
        </JumpList>
    </JumpList.JumpList>
</Application>
```

JumpList

Il y a également 2 events sur lesquels on peut s'abonner :
-JumpItemsRejected, obtient les raisons pour lesquelles on n'a pas pu ajouter l'item
-JumpItemsRemovedByUser

```
private void JumpList_JumpItemsRejected(object sender, System.Windows.Shell.JumpItemsRejectedEventArgs e)
{
    StringBuilder sb = new StringBuilder();
    sb.AppendFormat("{0} Jump Items Rejected:\n", e.RejectionReasons.Count);
    for (int i = 0; i < e.RejectionReasons.Count; ++i)
    {
        if (e.RejectedItems[i].GetType() == typeof(JumpPath))
            sb.AppendFormat("Reason: {0}\tItem: {1}\n", e.RejectionReasons[i], ((JumpPath)e.RejectedItems
[i]).Path);
        else
            sb.AppendFormat("Reason: {0}\tItem: {1}\n", e.RejectionReasons[i], ((JumpTask)e.RejectedItems
[i]).ApplicationPath);
    }

    MessageBox.Show(sb.ToString());
}
```

JumpList

Pour le 2ème event :

```
private void JumpList_JumpItemsRemovedByUser(object sender, System.Windows.Shell
.JumpItemsRemovedEventArgs e)
{
    StringBuilder sb = new StringBuilder();
    sb.AppendFormat("{0} Jump Items Removed by the user:\n", e.RemovedItems.Coun
t);
    for (int i = 0; i < e.RemovedItems.Count; ++i)
    {
        sb.AppendFormat("{0}\n", e.RemovedItems[i]);
    }

    MessageBox.Show(sb.ToString());
}
```

JumpList

On peut également ajouter ou supprimer des éléments au travers du code :

```
private void ClearJumpList(object sender, RoutedEventArgs e)
{
    //on récupère la jumplist courante de l'application
    JumpList jumpList1 = JumpList.GetJumpList(App.Current);
    //on supprime les éléments
    jumpList1.JumpItems.Clear();
    //on valide
    jumpList1.Apply();
}
```

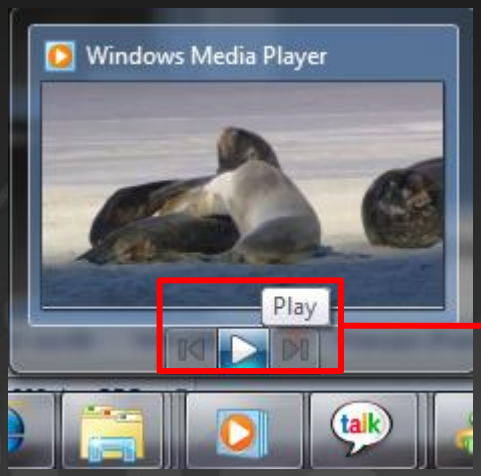
JumpList

On peut également ajouter ou supprimer des éléments au travers du code :

```
private void AddTask(object sender, RoutedEventArgs e)
    {
        //création d'une tacke
        JumpTask jumpTask1 = new JumpTask();
        //on renseigne les différents champs
        jumpTask1.ApplicationPath = System.IO.Path.Combine(Environment.GetFolderPath(Environment.SpecialF
    older.SystemX86), "calc.exe");
        jumpTask1.IconResourcePath = System.IO.Path.Combine(Environment.GetFolderPath(Environment.Special
    Folder.SystemX86), "calc.exe");
        jumpTask1.Title = "Calculatrice";
        jumpTask1.Description = "Ouvre la Calculatrice.";
        jumpTask1.CustomCategory = "User Added Tasks";
        //on récupère la liste courante
        JumpList jumpList1 = JumpList.GetJumpList(App.Current);
        //on ajoute l'élément
        jumpList1.JumpItems.Add(jumpTask1);
        JumpList.AddToRecentCategory(jumpTask1);
        //on applique la modification
        jumpList1.Apply();
    }
```

JumpList

- Les jumplists sont accessibles en faisant un clic droit sur l'icône en barre des tâches, mais si on survole ce dernier on peut insérer sur la miniature des boutons d'actions.
- Avec le media player par exemple :



On peut également ajouter ces boutons pour nos applications, à travers le code.

JumpList

Pour commencer on va rajouter des ressources :

```
<Window.Resources>
    <DrawingImage x:Key="PlayImage">
        <DrawingImage.Drawing>
            <DrawingGroup>
                <DrawingGroup.Children>
                    <GeometryDrawing Brush="Green" Geometry="F1 M 50,25L 0,0
L 0,50L 50,25 Z "/>
                </DrawingGroup.Children>
            </DrawingGroup>
        </DrawingImage.Drawing>
    </DrawingImage>
    <DrawingImage x:Key="StopImage">
        <DrawingImage.Drawing>
            <DrawingGroup>
                <DrawingGroup.Children>
                    <GeometryDrawing Brush="Gray" Geometry="F1 M 0,0L 50,0L
50,50L 0,50L 0,0 Z "/>
                </DrawingGroup.Children>
            </DrawingGroup>
        </DrawingImage.Drawing>
    </DrawingImage>
</Window.Resources>
```

JumpList

Il reste un dernier pré-requis à implémenter qui nous facilitera la vie pour la suite :

```
<Window.CommandBindings>
    <CommandBinding Command="MediaCommands.Play"
                    Executed="StartCommand_Executed"
                    CanExecute="StartCommand_CanExecute"/>
    <CommandBinding Command="MediaCommands.Stop"
                    Executed="StopCommand_Executed"
                    CanExecute="StopCommand_CanExecute"/>
</Window.CommandBindings>
```

Ceci va nous permettre d'appeler les commandes sur les autres boutons. De cette manière nous pourrons déclencher le même traitement sur plusieurs boutons

JumpList

Sur la fenêtre où on souhaite développer ces actions il faut rajouter ce code :

```
<Window.TaskbarItemInfo>
    <TaskbarItemInfo x:Name="taskBarItemInfo1"
                     Overlay="{StaticResource ResourceKey=StopImage}"
                     ThumbnailClipMargin="80,0,80,140"
                     Description="Taskbar Item Info Sample" >
    </TaskbarItemInfo>
</Window.TaskbarItemInfo>
```

Permet d'indiquer que l'on va disposer des boutons sur la miniature.

L'overlay ici permet de rajouter une image directement sur l'icône dans la barre des taches

JumpList

Rajout des boutons :

```
<Window.TaskbarItemInfo>
    <TaskbarItemInfo x:Name="taskBarItemInfo1"
                     Overlay="{StaticResource ResourceKey=StopImage}"
                     ThumbnailClipMargin="80,0,80,140"
                     Description="Taskbar Item Info Sample">
        <TaskbarItemInfo.ThumbButtonInfos>
            <ThumbButtonInfoCollection>
                <ThumbButtonInfo
                    DismissWhenClicked="False"
                    Command="MediaCommands.Play"
                    CommandTarget="{Binding ElementName=btnPlay}"
                    Description="Play"
                    ImageSource="{StaticResource ResourceKey=PlayImage}"/>
                <ThumbButtonInfo
                    DismissWhenClicked="True"
                    Command="MediaCommands.Stop"
                    CommandTarget="{Binding ElementName=btnStop}"
                    Description="Stop"
                    ImageSource="{StaticResource ResourceKey=StopImage}"/>
            </ThumbButtonInfoCollection>
        </TaskbarItemInfo.ThumbButtonInfos>
    </TaskbarItemInfo>
</Window.TaskbarItemInfo>
```

JumpList

- La propriété dismissWhenClicked introduite dans le slide précédent permet de faire disparaître ou non la miniature lors du clic sur ce bouton.

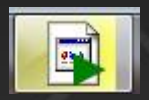
JumpList

Lors de l'appui sur le bouton start, l'action suivante sera déclenchée :

```
private void StartCommand_Executed(object sender, ExecutedRoutedEventArgs e)
{
    if (this._backgroundWorker.IsBusy == false)
    {
        this._backgroundWorker.RunWorkerAsync();

        this.taskBarItemInfo1.ProgressState = TaskbarItemProgressState.Normal;
        this.taskBarItemInfo1.Overlay = (DrawingImage)this.FindResource("PlayImage");
    }
    e.Handled = true;
}
```

Normal



Pause

Erreur

Intermédiaire

Permet de changer la couleur de la progress bar

Les boutons peuvent déclencher n'importe quelle action, ici cela permet d'introduire un nouvel effet disponible sur Seven

JumpList

Lors de l'appui sur le bouton stop, l'action suivante sera déclenchée :

```
private void StopCommand_Executed(object sender,
 ExecutedRoutedEventArgs e)
{
    this._backgroundWorker.CancelAsync();
    e.Handled = true;
}
```

Input Binding

Input Binding

- Un inputBinding représente une liaison entre un inputGesture et une commande.
- InputGesture permet de décrire les événements des périphériques d'entrée
  - Souris
  - clavier

Input Binding

- A quoi cela peut il servir ?
  - Rajouter des touches d'aide (F1)
  - Détecter les combinaisons de touches (ctr/alt/maj + …)
  - Les clics de souris
  - Les clics de souris combinés avec des touches claviers.

Input Binding

La syntaxe de base se traduit en xaml par :

```
<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open"
 Executed="CommandBindingOpen_Executed" />
</Window.CommandBindings>
<Window.InputBindings>
    <KeyBinding Key="B"
          Modifiers="Control"
          Command="ApplicationCommands.Open"  />
</Window.InputBindings>
```

Lorsque sur l'application on effectuera un ctrl + b alors l'action définie dans CommandBindingOpen_Executed sera déclenchée

Input Binding

Maintenant si l'on souhaite déclencher une commande personnalisée, il faut implémenter cette classe :

```
public class TestInputCommand : ICommand
{
    public Key GestureKey { get; set; }
    public ModifierKeys GestureModifier { get; set; }
    public MouseAction MouseGesture { get; set; }

    Action<object> _executeDelegate;

    public TestInputCommand(Action<object> executeDelegate)
    {
        _executeDelegate = executeDelegate;
    }

    public void Execute(object parameter)
    {
        _executeDelegate(parameter);
    }

    public bool CanExecute(object parameter) { return true; }
    public event EventHandler CanExecuteChanged;
}
```

Classe de base fournie par la msdn.

Input Binding

Au niveau de la fenêtre où vous devez implémenter votre input binding, il faut le déclarer :

```
public TestInputCommand ChangeColorCommand
{
    get { return _changeColorCommand; }
}

private TestInputCommand _changeColorCommand;
```

Puis l'initialiser par exemple dans le constructeur :

```
DataContext = this;

_changeColorCommand = new TestInputCommand(x => this.ChangeColor(x));
_changeColorCommand.GestureKey = Key.C;
_changeColorCommand.GestureModifier = ModifierKeys.Control;
ChangeColorCommand.MouseGesture = MouseAction.RightClick;
```

Il ne reste qu'à coder la fonction ChangeColor pour y mettre votre action sur l'interface.

Input Binding

Pour la partie XAML :

```
<StackPanel Background="Transparent">
    <StackPanel.InputBindings>
      <KeyBinding Command="{Binding ChangeColorCommand}"
      CommandParameter="{Binding ElementName=colorPicker, Path=SelectedItem}"
      Key="{Binding ChangeColorCommand.GestureKey}"
      Modifiers="{Binding ChangeColorCommand.GestureModifier}" />

      <MouseBinding Command="{Binding ChangeColorCommand}"
        CommandParameter="{Binding ElementName=colorPicker, Path=SelectedItem}"
        MouseAction="{Binding ChangeColorCommand.MouseGesture}" />
    </StackPanel.InputBindings>

    <Button Content="Change Color"
  Command="{Binding ChangeColorCommand}"
  CommandParameter="{Binding ElementName=colorPicker, Path=SelectedItem}">
    </Button>

    <ListBox Name="colorPicker"
   Background="Transparent"
   xmlns:sys="clr-namespace:System;assembly=mscorlib">
        <sys:String>Red</sys:String>
        <sys:String>Green</sys:String>
    </ListBox>
</StackPanel>
```

Input Binding

La déclaration peut être simplifiée et faciliter la sérialisation d'un composant :

```
<KeyBinding Command="{Binding ChangeColorCommand}"
CommandParameter="{Binding ElementName=colorPicker, Path=SelectedItem}"
Key="{Binding ChangeColorCommand.GestureKey}"
Modifiers="{Binding ChangeColorCommand.GestureModifier}" />
```

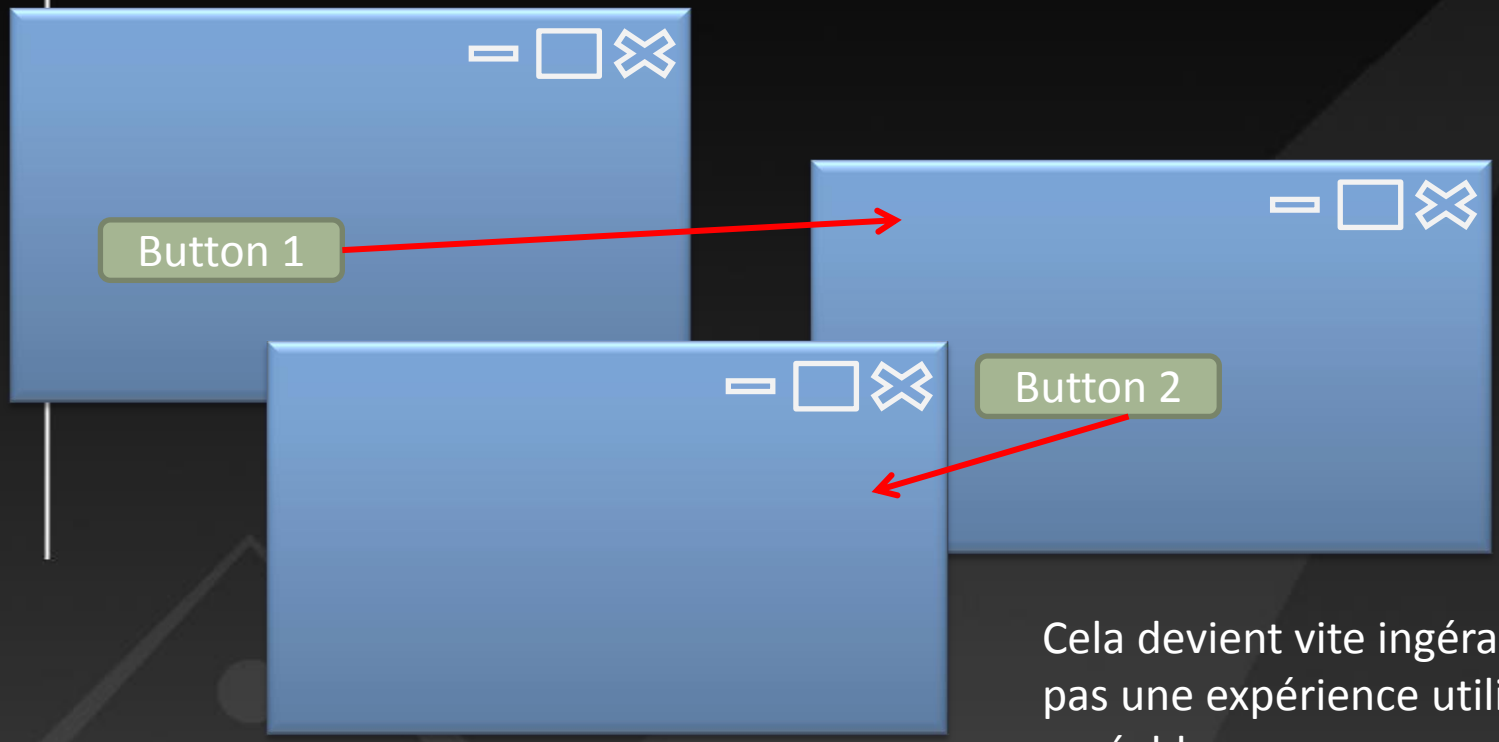
La touche ainsi que la combinaison peuvent être stockées dans une seule propriété :

```
<KeyBinding Command="{Binding ChangeColorCommand}"
CommandParameter="{Binding ElementName=colorPicker, Path=SelectedItem}"
Gesture="CTRL+C"/>
```

PRINCIPE D'UNE INTERFACE

Principe d'une interface

- Lors des 1ères applications, la tentation de réaliser des applications multi fenêtres reste forte.



Cela devient vite ingérable et n'offre pas une expérience utilisateur agréable

Principe d'une interface

- L'utilisation du pattern MVVM ne vous évitera pas non plus ce piège.
- Vous devez donc penser vos applications de telle manière qu'il y ait un minimum de fenêtres et que toutes les informations soient disponibles rapidement.
- Exemple :

Main.xaml avec une grille
1ère ligne ➔ le menu
2 ème ligne ➔ le contenu

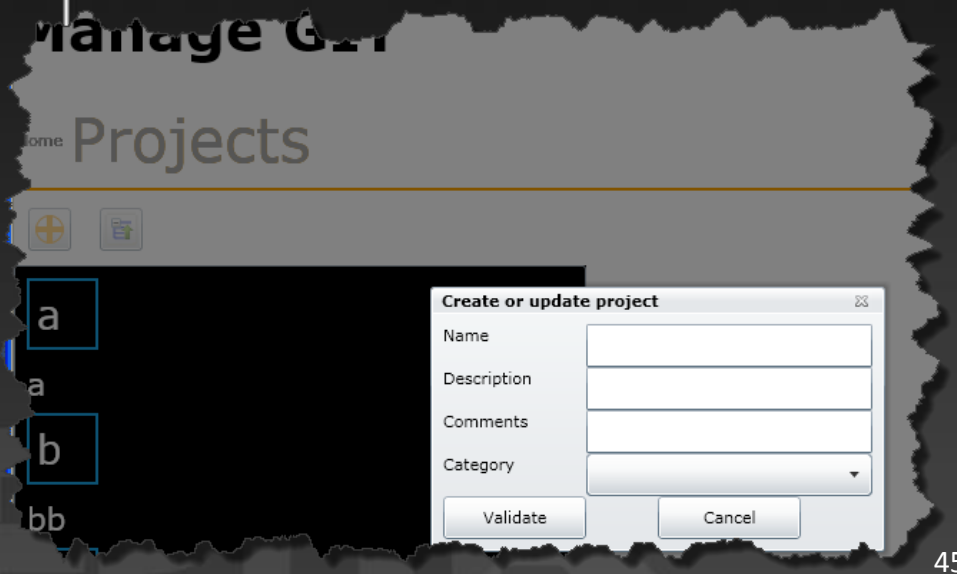
Le menu possède des commandes dans le view Model du Main.xaml.
Ces commandes manipulent le contenu à l'aide d'une variable bindée sur l'élément de la vue

Le contenu peut être un user Contrôle (dont le viewModel sera identique au Main.xaml ou alors une nouvelle vue avec un nouveau ViewModel.
Les transitions peuvent se déclencher sur le changement du contenu.

N'oubliez pas de bien découper votre interface !

Principe d'une interface

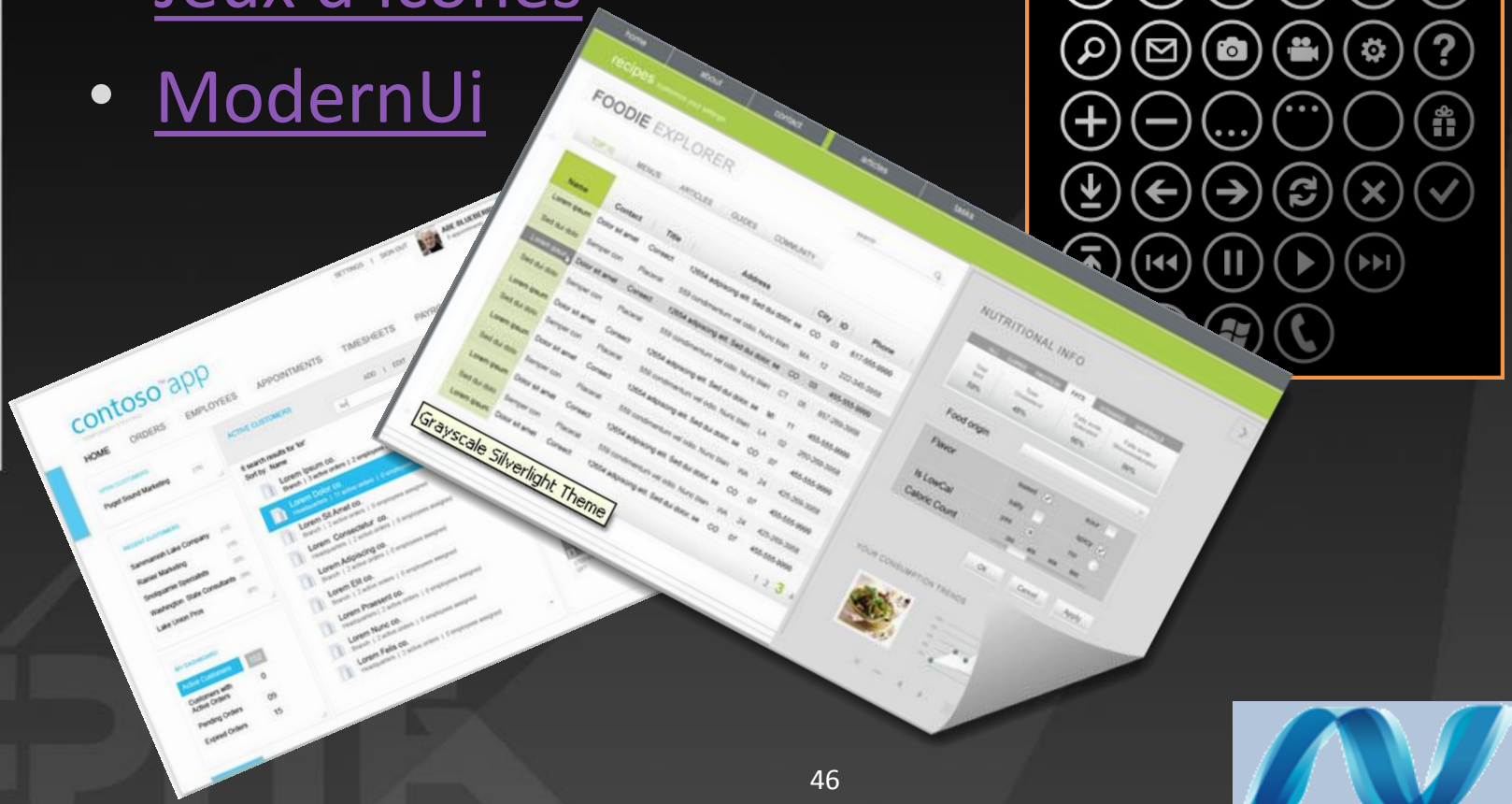
- Pour les ajouts / modifications, penser :
  - Aux fenêtres modales ceci évite d'ouvrir une nouvelle fenêtre
  - A l'utilisation du VSM (voir les styles)



Attention, il ne faut pas non plus abuser des animations.

Principe d'une interface

- Jeux d'icônes
- Jeux d'icônes
- ModernUi



WPF 4.5

WPF 4.5

- Validation de données Asynchrone
- Nouvelles méthodes sur le dispatcher
- Délais sur les bindings (évite les calculs lors de changement rapide)
- Binding sur propriété static
- Live shaping (filtres, tris, groupes)
- Nouvelles fonctionnalités de virtualisations
- …

Autre

Autre

- Nous avons abordé durant le cours une version légère du MVVM répondant à une petite partie des problématiques. Microsoft a publié un ensemble de bonnes pratiques afin de construire une application modulable sous WPF / Silverlight. Cette publication est connue sous le nom de PRISM

Présentation de PRISM

Microsoft®
patterns & practices
proven practices for predictable results

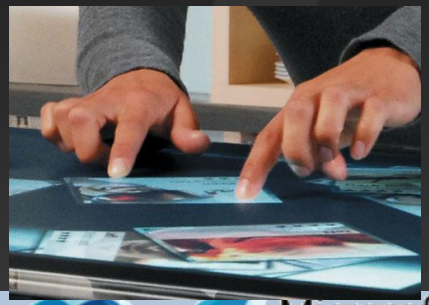
Exemple utilisant PRISM

Microsoft®
.NET

Autre

- Une dernière partie que nous n'avons pas abordée est l'implémentation d'interfaces tactiles au travers de WPF :

Exemple d'implémentation



Questions ?