

# WPF

---

# WPF

# Sommaire

1. Positionnement de WPF
2. Création d'un projet WPF
3. Concepts xml
4. Binding
5. Styles et positionnement
6. Composant graphique
7. Principaux patterns et idiomes

# 1. Positionnement de WPF

## Découverte de WPF

- **WPF (Windows Presentation Foundation)** : Framework pour le développement d'applications desktop sous Windows.
- **Avantages** : Interface riche, utilisation de xml, séparation entre l'interface et la logique métier.
- **Architecture** : Séparation claire entre l'interface utilisateur (xml) et la logique métier (C#).
- Développé par Microsoft, lancé avec .NET Framework 3.0 en **2006**
- Devenu **open-source** sous licence MIT en **2018**

## Windows Forms / WPF / UWP

- **Windows Forms** : Ancien, simple à utiliser, mais limité pour les interfaces avancées.
- **WPF** : Moderne, personnalisable, orienté UI riche, supporte MVVM, utilisation de DirectX pour les graphiques et les animations.
- **UWP (Universal Windows Platform)** : Pour applications modernes sur Windows 10/11, tablettes et Xbox. Optimisé pour le multiplateformes, plus performant mais avec une personnalisation moins évoluée que WPF.

## MAUI / Xamarin

- **Xamarin** : Framework mature pour créer des applications modernes et performantes pour iOS, Android et Windows. (fin du support le 1er Mai 2024)
- **MAUI (Multi-platform App UI)** : Successeur de Xamarin.Forms, unifie le développement d'applications pour plusieurs plateformes (iOS, Android, Windows, etc.). Optimisé grâce à .NET 6 offrant de meilleures performances et une réactivité accrue.

## Alternative à WPF : Avalonia et Uno

- **Avalonia UI** : Framework multi-plateforme pour des applications de bureau (Windows, macOS, Linux) inspiré par WPF et UWP. Avalonia utilise le moteur Skia Graphics pour son rendu multi-plateforme.
- **Uno Platform** : Framework multi-plateforme pour des applications natives (iOS, Android, Windows, Web) en C#. Se rapproche plus de .NET MAUI.



## WPF et Windows 10 & 11

- **Compatibilité** : WPF est pleinement compatible avec Windows 10 et 11.
- **Nouvelles fonctionnalités** : Support pour les API Windows modernes, meilleure performance graphique.

## Avantages de WPF

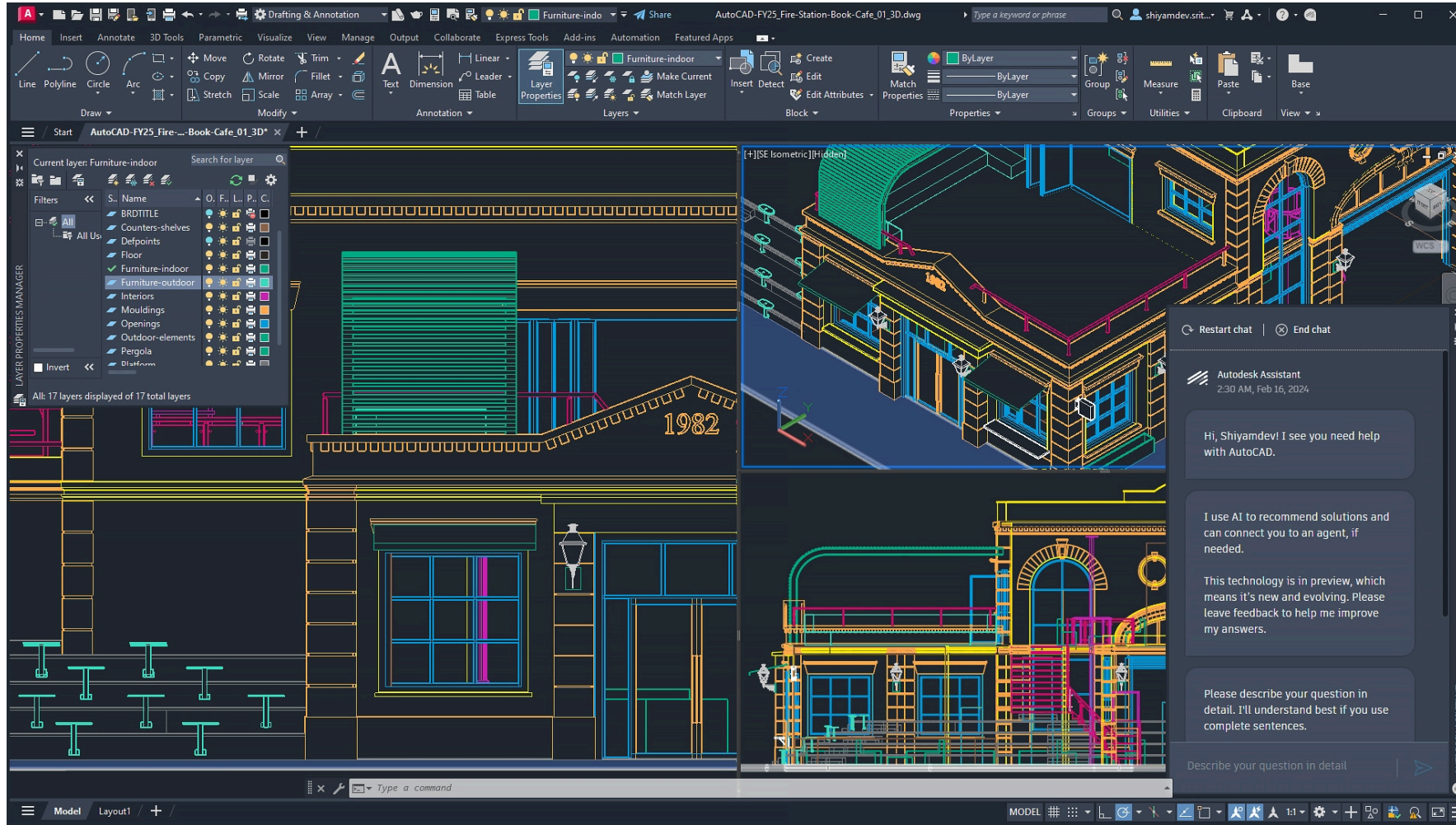
- **Interface Utilisateur Riche** : Graphiques 2D/3D, animations, multimédias.
- **Séparation Logique/Interface** : Utilise xml pour une meilleure maintenance.
- **Liaison de Données Avancée** : Liaison flexible et puissante.
- **Personnalisation et Thématisation** : Contrôles et styles hautement personnalisables.
- **Support Matériel** : Accélération matérielle via DirectX pour de meilleures performances.

## Applications réalisées avec WPF

Plusieurs grandes entreprises ont utilisées le WPF pour mettre au point leurs solutions logicielles, on peut notamment citer :

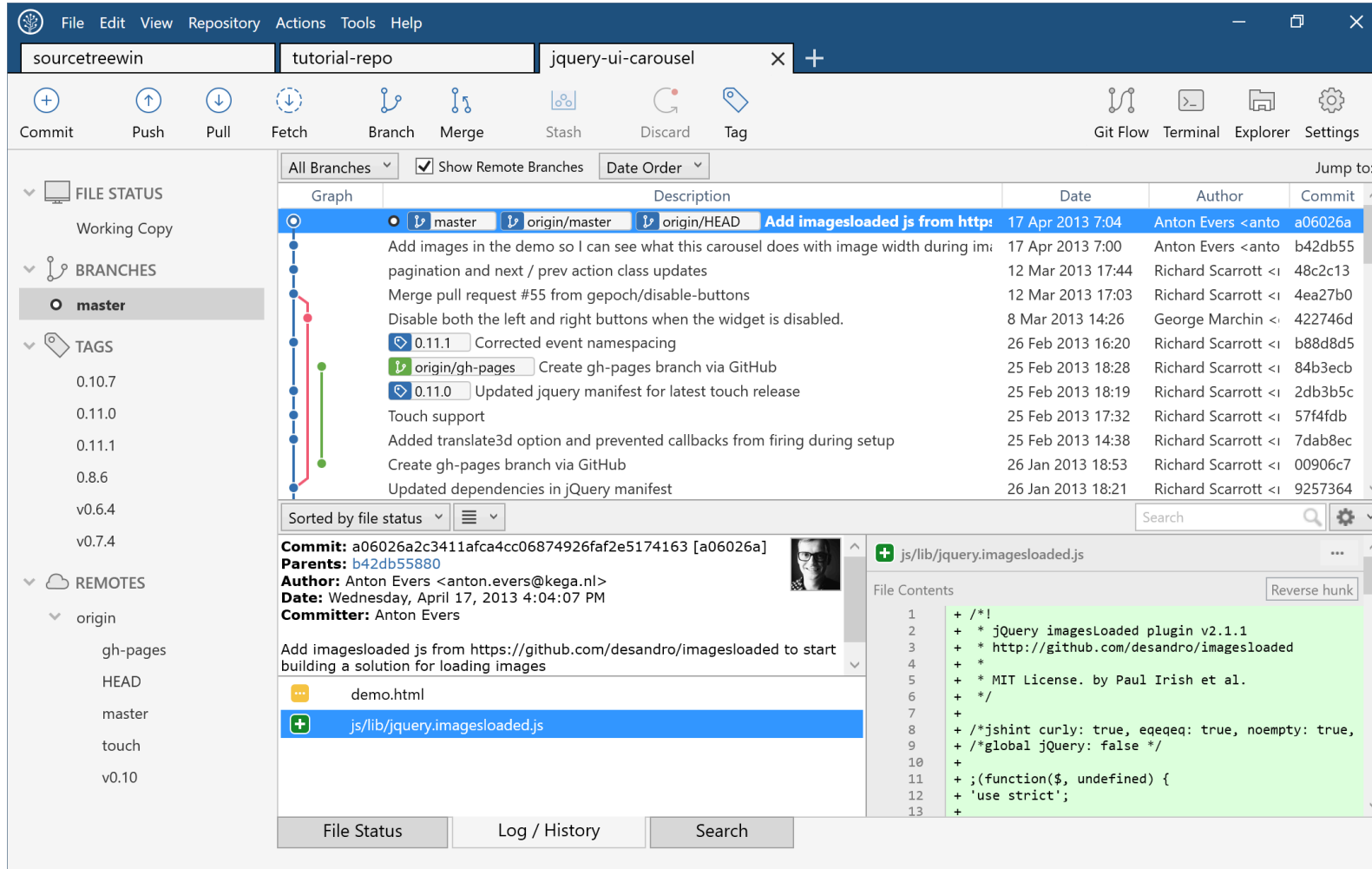
- **AutoCAD** : Logiciel de conception assistée par ordinateur (CAO) d'Autodesk
- **SQL Server Management Studio** - Outil de gestion pour Microsoft SQL Server
- **SourceTree** - Client graphique pour git créé par Atlassian

# AutoCAD





# SourceTree

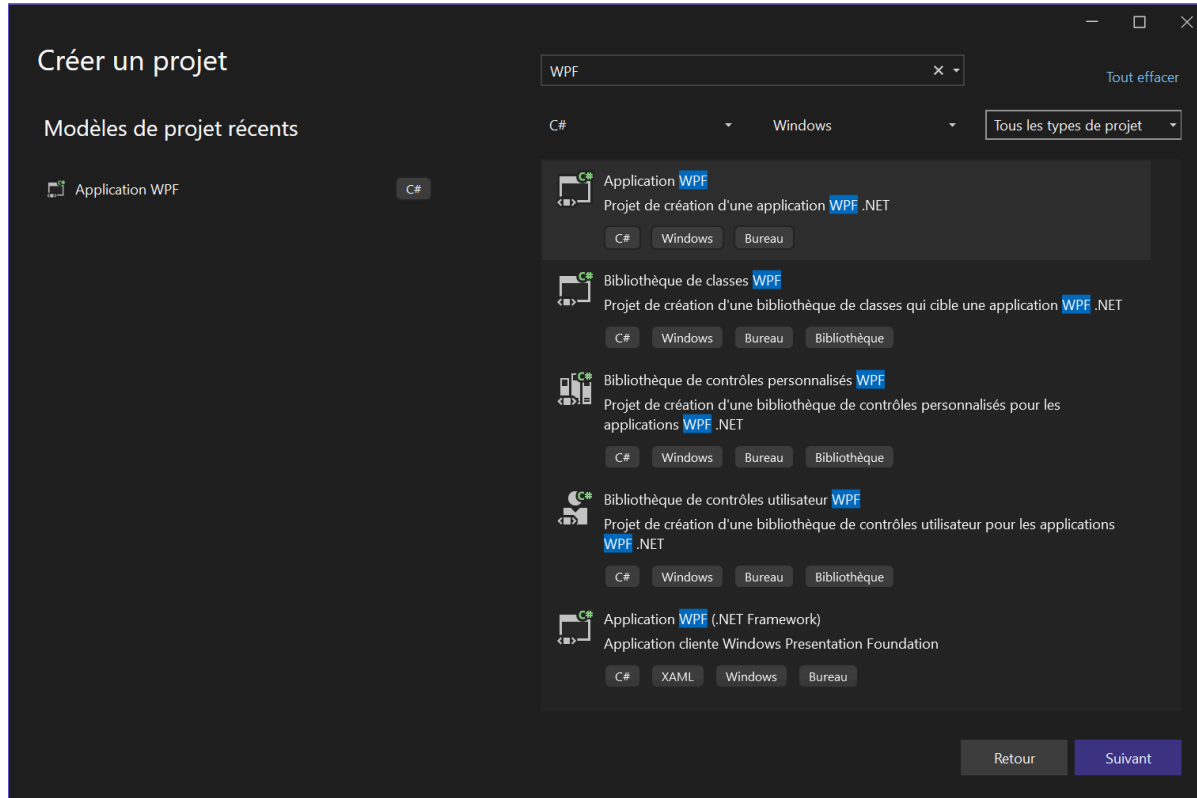


The screenshot shows the SourceTree application interface. The top menu bar includes File, Edit, View, Repository, Actions, Tools, and Help. Below the menu is a toolbar with icons for Commit, Push, Pull, Fetch, Branch, Merge, Stash, Discard, and Tag. The main window is divided into several panes:

- Left Pane:** Contains 'FILE STATUS' (Working Copy), 'BRANCHES' (listing 'master'), 'TAGS' (listing versions like 0.10.7, 0.11.0, etc.), and 'REMOTES' (listing 'origin' and its branches).
- Top Right Pane:** Shows a commit history table with columns for Graph, Description, Date, Author, and Commit. The current commit is 'a06026a' by Anton Evers, dated 17 Apr 2013 7:04, with the description 'Add imagesloaded.js from https://github.com/desandro/imagesloaded to start building a solution for loading images'.
- Bottom Left Pane:** Shows the commit details for 'a06026a', including the author (Anton Evers), date (Wednesday, April 17, 2013 4:04:07 PM), and a list of files (demo.html, js/lib/jquery.imagesloaded.js).
- Bottom Right Pane:** Shows the file content for 'js/lib/jquery.imagesloaded.js', displaying the JavaScript code for the 'jQuery imagesLoaded plugin v2.1.1'.

## 2. Création d'un projet WPF

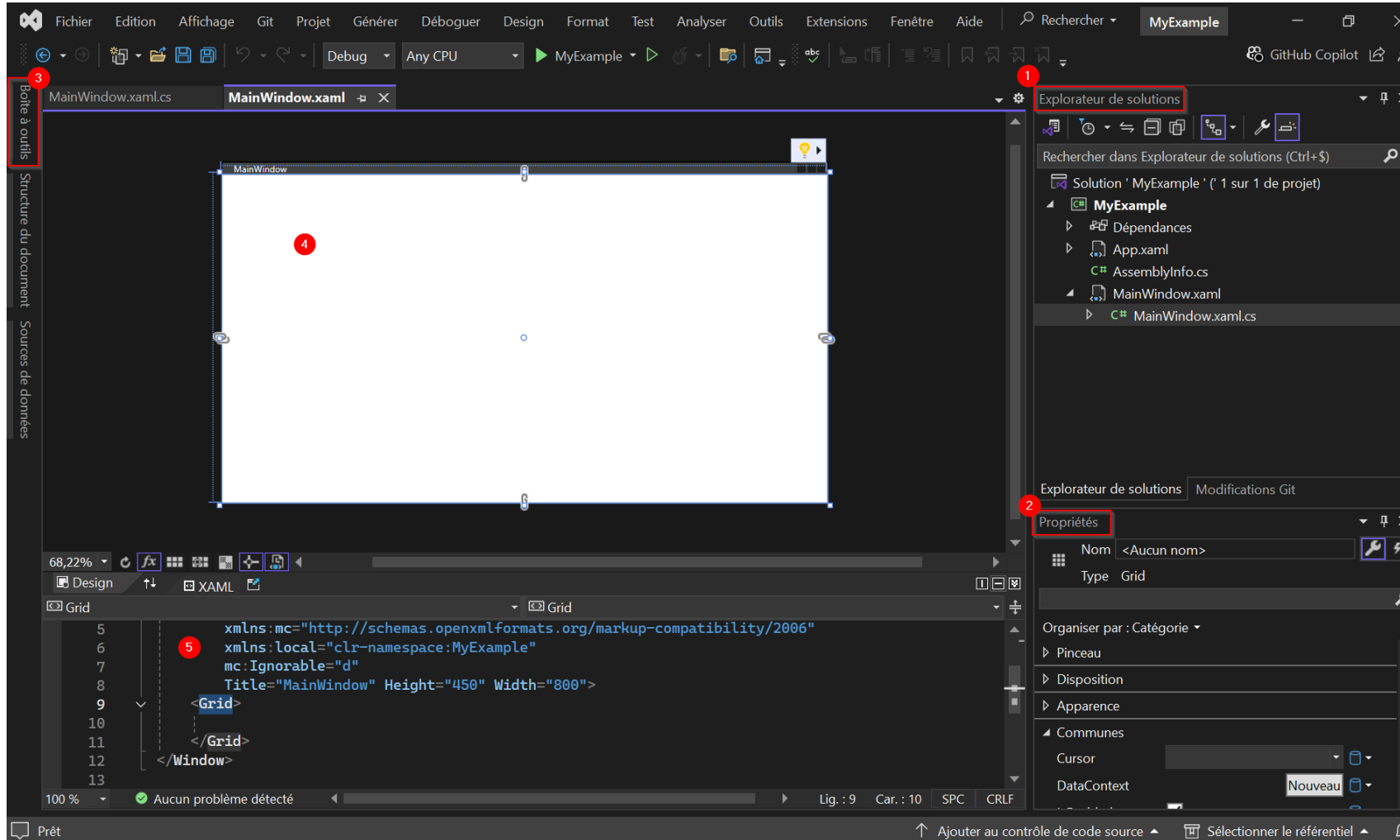
# Initialisation du projet



1. Ouvrir Visual Studio et cliquer sur : "**Créer un projet**"
2. Saisir : "**WPF**" dans la recherche de modèles
3. Sélectionner: "**Application WPF**"



# Interface de Visual Studio 2022



## Description des éléments (1/2)

1. **Explorateur de solutions** : Tous les fichiers de projet, code, fenêtres et ressources apparaissent dans ce volet.
2. **Propriétés** : Affiche les paramètres configurables en fonction de l'élément sélectionné (fichier, objet dans le Designer, etc.).
3. **Boîte à outils** : Contient tous les contrôles que vous pouvez ajouter à un formulaire. Ajoutez un contrôle en double-cliquant ou en le glissant-déposant.

## Description des éléments (2/2)

4. **Concepteur xml** : Concepteur interactif pour les documents xml. Permet de composer visuellement l'interface utilisateur en glissant-déposant des objets depuis la Boîte à outils. Les modifications sont synchronisées avec l'éditeur de code.
5. **Editeur de code xml** : Éditeur de code pour les documents xml. Permet de créer l'interface utilisateur manuellement. Les modifications sont synchronisées avec le concepteur. Les propriétés de l'objet sélectionné sont affichées dans le volet Propriétés.

# Structure du projet

```
AppWPF
├── App.xml
│   └── App.xml.cs
├── AssemblyInfo.cs
└── MainWindow.xml
    └── MainWindow.xml.cs
```

- **App.xml**: point d'entrée déclaratif de l'application, contient les ressources globales et le fichier de démarrage de l'application ( `StartupUri` )
- **AssemblyInfo.cs**: Contient des informations sur l'assembly, telles que le nom, la version et d'autres attributs de l'assembly.
- **MainWindow.xml**: définit l'interface utilisateur principale de l'application.

## 2. Concepts xml

# Définition du xml

- **xml (eXtensible Application Markup Language)** : Langage de balisage utilisé pour définir les interfaces utilisateur en WPF.
- **Structure** : Hiérarchique, permet de décrire l'UI de manière déclarative.
- **Fonctionnalité** : Permet de séparer la conception de l'interface utilisateur de la logique métier en utilisant des fichiers xml pour la mise en page et des fichiers code-behind pour la logique.

# Glossaire

Terme	Définition
<b>Contrôle</b>	Un composant de l'interface utilisateur, comme un bouton ou une fenêtre
<b>Attribut</b>	Propriété d'un élément définie directement dans la balise de l'élément, par exemple, <code>&lt;Button Content="Cliquez ici" /&gt;</code>
<b>Code-behind</b>	Fichier C# associé à un fichier xml, contenant la logique de l'application
<b>Binding</b>	Liaison de données entre l'interface utilisateur et les sources de données, permettant une mise à jour automatique de l'interface
<b>Resource</b>	Définition réutilisable de styles, de modèles ou d'autres objets, souvent stockée dans un dictionnaire de ressources
<b>Template</b>	Modèle définissant l'apparence et la structure d'un contrôle

# Glossaire

Terme	Définition
<b>Resource</b>	Définition réutilisable de styles, de modèles ou d'autres objets, souvent stockée dans un dictionnaire de ressources
<b>Template</b>	Modèle définissant l'apparence et la structure d'un contrôle
<b>DataContext</b>	Objet de données auquel les éléments de l'interface utilisateur sont liés pour les opérations de binding
<b>Dependency Property</b>	Propriété qui prend en charge le système de propriétés de dépendance de WPF, permettant des fonctionnalités avancées comme le data binding, les animations et les styles



# Glossaire

Terme	Définition
<b>Routed Event</b>	Événement qui peut être géré par plusieurs éléments dans une hiérarchie d'éléments, permettant une propagation des événements à travers l'arborescence visuelle
<b>Storyboard</b>	Conteneur pour les animations, permettant de définir et de contrôler des séquences d'animations complexes
<b>Trigger</b>	Mécanisme permettant de modifier les propriétés d'un contrôle en réponse à des événements ou des changements de propriété
<b>Visual Tree</b>	Représentation hiérarchique de tous les éléments visuels dans une application WPF, utilisée pour le rendu et la gestion des événements

# Représentation du xml

- Le langage xml est généralement stocké dans un fichier avec l'extension: `.xml`
- Ces fichiers sont généralement encodés au format `UTF-8`
- Voici un exemple de représentation d'un bouton en xml:

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

## Elements qui composent un objet xml

- **Un objet xml** est une **instances d'un type**, ce type est défini dans les assembly de la technologie xml.
- Les objets sont rédigés en **PascalCase** : `StackPanel1`, `Button`, ...

## Déclaration d'objet

Dans l'exemple précédent la balise `<Button/>` était auto-fermante, cette syntaxe est valable lorsque l'objet ne comporte aucun contenu. Autrement il faudrait utiliser la syntaxe suivante:

```
<StackPanel>  
    <Button>Hello world!</Button>  
</StackPanel>
```

Lorsque l'on spécifie un objet xml, il est instancié avec son constructeur sans paramètres.

## Syntaxe d'attributs (propriétés)

- Les propriétés d'un objet peuvent être exprimée à l'aide des attributs de l'élément.
- La syntaxe d'un attribut est la suivante: `Attribut="Valeur"`
- La valeur de l'attribut est toujours spécifiée sous forme de chaîne à l'aide de guillemets.

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

# Syntaxe des éléments d'une propriété

- Pour certaines propriétés d'un objet xml la syntaxe par attribut n'est pas possible.
- Pour cela une syntaxe spécifique est utilisée: `<Type.Propriété>`
- Généralement, le contenu de cet élément contient un objet qui permet de spécifier la propriété

```
<Button>  
  <Button.Background>  
    <SolidColorBrush Color="Blue" />  
  </Button.Background>  
</Button>
```

## Syntaxe pour les collections

xml fournit certaines optimisations pour produire du code plus lisible. Si une propriété est de type `Collection`, alors les éléments enfants de cette propriétés feront partie de cette collection.

```
<LinearGradientBrush>
  <LinearGradientBrush.GradientStops>
    <!-- pas de nouvelle collection explicite GradientStopCollection,
    l'analyseur sait comment la trouver ou la créer -->
    <GradientStop Offset="0.0" Color="Red" />
    <GradientStop Offset="1.0" Color="Blue" />
  </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

## Propriétés du contenu xml

Le xml permet à une classe de spécifier laquelle de ses propriétés peut-être considérée comme son *contenu* xml (l'élément enfant).

Pour ce type d'élément xml, il n'y a pas besoin de préciser l'attribut de l'élément, ce qui rend la relation parent enfant plus lisible.

```
<Border>  
    <TextBox Width="300" />  
</Border>
```

```
<Border>  
    <Border.Child>  
        <TextBox Width="300" />  
    </Border.Child>  
</Border>
```



## Contenu sous forme de texte

Un petit nombre d'éléments xml peuvent traiter directement du texte en tant que contenu. Pour cela il faut soit être :

1. Une classe qui défini une propriété de contenu qui peut être assigné en tant que chaîne (exmples `Object`)
2. Le type doit avoir un convertisseur de type
  - `<Brush>Blue</Brush>` : converti la valeur en brosse bleue
3. Le type doit être une primitive connue du langage xml.

# Propriétés de contenu et collections combinés

- Dans l'exemple de gauche on peut omettre la propriété `Children` qui est la propriété de contenu de l'élément `StackPanel`.
- La propriété `Children` est de type `UIElementCollection` qui implémente `IList`, on peut donc utiliser la syntaxe de collection.

```
<!-- Version simplifiée ! -->
<StackPanel>
    <Button>First Button</Button>
    <Button>Second Button</Button>
</StackPanel>
```

```
<StackPanel>
    <StackPanel.Children>
        <!--<UIElementCollection>-->
        <Button>First Button</Button>
        <Button>Second Button</Button>
        <!--</UIElementCollection>-->
    </StackPanel.Children>
</StackPanel>
```

## Déclaration d'événements dans les attributs

La définition des attributs peut également être utilisée pour des membres qui sont des **événements** plutôt que des propriétés.

Le nom de l'attribut est le **type d'événement** et sa valeur **le nom de la méthode déléguée** qui gère cet événement.

```
<Button Click="Button_Click" >Click Me!</Button>
```

## Sensibilité à la casse en xml

- xml **est sensible à la casse** pour les types, éléments d'objet, éléments de propriété et noms d'attributs.
- Les **mots-clés** et **primitives** du langage xml sont également sensibles à la casse.
- **Les valeurs** peuvent ne pas être sensibles à la casse, selon le convertisseur de type.

## Gestion des espaces blanc en xml

- Les processeurs WPF xml ignorent ou normalisent les espaces blancs non significatifs.
- Les espaces, sauts de ligne et tabulations sont convertis en espaces.
- Un seul espace est conservé aux extrémités d'une chaîne de texte.

# Extension de balises

- **Extensions de balisage en xml :**
  - Utilisent des accolades `{ }` pour indiquer leur usage.
  - Permettent de traiter les valeurs d'attributs autrement que comme des chaînes littérales.
- **Extensions courantes en WPF :**
  - `Binding` : Pour les liaisons de données.
  - `StaticResource` : Référence une ressource statique.
  - `DynamicResource` : Référence une ressource dynamique.

## Fonctionnalités des extensions de balise

- Attribuer des valeurs à des propriétés sans syntaxe d'attributs.
- Différer les valeurs ou référencer des objets présents à l'exécution.

La liste de toutes les extensions de balises est disponible ici :

- [WPF xml Extensions](#)

## Exemple

`StaticResource` définit la propriété `Style` en référence à une ressource de style dans un dictionnaire de ressources.

```
<Window.Resources>
  <SolidColorBrush x:Key="MyBrush" Color="Gold" />
  <Style TargetType="Border" x:Key="PageBackground">
    <Setter Property="BorderBrush" Value="Blue" />
    <Setter Property="BorderThickness" Value="5" />
  </Style>
</Window.Resources>
<Border Style="{StaticResource PageBackground}">
  <StackPanel>
    <TextBlock Text="Hello" />
  </StackPanel>
</Border>
```



## Conversion de types

En xml les valeurs des attributs doivent être définies par des chaînes. La conversion de chaînes en objets ou valeur primitives est gérée nativement par la plupart des objets xml.

Exemple avec **Thickness** qui est un type utilisée pour les propriétés comme **Margin**:

```
<Button Margin="10,20,10,30" Content="Click me" />
```

```
<Button Content="Click me">  
  <Button.Margin>  
    <Thickness Left="10" Top="20" Right="10" Bottom="30" />  
  </Button.Margin>  
</Button>
```

## Elements racine

- Un fichier xml doit avoir un **seul élément racine** pour être un fichier XML bien formé et un fichier xml valide.
- L'élément racine a une signification importante en WPF :
  - `Window` pour une fenêtre ou `Page` pour des pages
  - `ResourceDictionary` pour un dictionnaire externe
  - `Application` pour la définition de l'application

## Exemple de code d'un élément racine

Voici un exemple d'élément racine pour une page WPF :

```
<Page x:Class="index.Page1"  
      xmlns="http://schemas.microsoft.com/winfx/2006/xml/presentation"  
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xml"  
      Title="Page1">  
</Page>
```

## Les namespaces (espaces de nom)

L'élément racine contient aussi les attributs `xmlns` et `xmlns:x`.

Ces attributs indiquent au processeur xml les espaces de noms xml qui contiennent les définitions de types que le balisage va référencer.

- `xmlns` : Définit l'espace de noms xml par défaut, permet d'utiliser les éléments WPF sans préfixe
- `xmlns:x` : Définit un espace de noms xml supplémentaire, permet d'utiliser des fonctionnalités comme `x:Class`, `x:Name`, `x:Key`, etc

## Namespace à partir d'un assembly personnalisé

Le mappage aux classes et assemblies personnalisés dans xml permet d'étendre les fonctionnalités de base en intégrant des types définis par l'utilisateur. Cela permet de :

1. **Utiliser des types personnalisés** : utiliser des classes et contrôle définit par l'utilisateur
2. **Avoir un code modulaire** : faciliter la réutilisation des éléments entre différents projets
3. **Interopérabilité** : ajouter des composants de libraries tierces

## Le préfixe `x:`

`x:` est un raccourci de l'espace de nom précédemment importé.

- `x:Key` : Définit une clé unique pour les ressources dans un `ResourceDictionary`.
- `x:Class` : Spécifie le namespace et la classe pour le code-behind.
- `x>Name` : Attribue un nom d'objet pour l'exécution.
- `x:Static` : Référence une valeur statique qui n'est pas autrement compatible avec xml.
- `x:Type` : Construit une référence de type. Utilisé pour spécifier des attributs qui prennent un type, comme `Style.TargetType`.

## Préfixes et types personnalisés

Pour les assembly personnalisés, il suffit de mapper un `xmlns` suivi du nom du préfixe souhaité.

On peut ensuite utiliser les contrôles présent dans l'assembly avec la syntaxe `<prefix:Control>`

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xml"
  xmlns:custom="clr-namespace:NumericUpDownCustomControl;assembly=CustomLibrary">
  <StackPanel Name="LayoutRoot">
    <custom:NumericUpDown Name="numericCtrl1" Width="100" Height="60"/>
  </StackPanel>
</Page>
```

# Événements et Code-Behind

- Les applications WPF utilisent xml pour le balisage et Visual Basic ou C# pour le code-behind.
- Le fichier xml est lié au code-behind via l'attribut `x:Class`.
- Pour ajouter un comportement à un élément, on utilise des événements et des gestionnaires d'événements.
- Le nom de l'événement et du gestionnaire sont spécifiés dans le xml.
- Le code du gestionnaire est défini dans le code-behind.



# xml dans Visual Studio

- **Éditeur xml** : Outil intégré dans Visual Studio pour créer et modifier les fichiers xml.
- **Design View et Code View** : Visualisation en temps réel de l'interface, possibilité de modifier directement le code xml.
- **Debugging** : Outils pour tester et déboguer l'interface utilisateur.

# Exemple événements et code-behind

## Code xml

```
<Page x:Class="ExampleNamespace.ExamplePage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xml">
  <StackPanel>
    <Button Click="Button_Click">Click me</Button>
  </StackPanel>
</Page>
```

## Code-behind

```
public partial class ExamplePage : Page
{
    public ExamplePage() =>
        InitializeComponent();

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        var buttonControl = (Button)e.Source;
        buttonControl.Foreground = Brushes.Red;
    }
}
```

## Routed events

- **Définition** : Un routed event permet à un élément de gérer un événement déclenché par un autre élément, à condition qu'ils soient connectés par une relation d'arbre (comme un parent et un enfant dans une interface utilisateur).
- **Propagation** : Les événements routés peuvent se propager à travers l'arbre visuel de l'application, ce qui signifie qu'ils peuvent être capturés et gérés par des éléments parents ou enfants

# Types de propagations

1. **Bubbling** : L'événement se propage des éléments enfants vers les éléments parents. Par exemple, un clic sur un bouton peut être capturé par le bouton lui-même, puis par son conteneur parent, et ainsi de suite jusqu'à la racine.
2. **Tunneling** : L'événement se propage des éléments parents vers les éléments enfants. Cela permet aux éléments parents de capturer l'événement avant qu'il n'atteigne l'élément cible.
3. **Direct** : L'événement est géré uniquement par l'élément qui l'a déclenché, sans propagation

## Exemple de routed event

Dans cet exemple, on capture l'événement `Click` de l'élément `Button` au niveau de son parent `StackPanel`

```
<StackPanel Button.Click="Button_Click">  
    <Button Content="Click me" />  
</StackPanel>
```

```
private void Button_Click(object sender, RoutedEventArgs e)  
{  
    // Logique de gestion de l'événement  
}
```

L'avantage de cette approche est qu'elle facilite la réutilisation du gestionnaire d'événements pour plusieurs éléments.

## Named element

Un élément nommé en xml est un élément auquel on a attribué un identifiant unique à l'aide de l'attribut `x:Name`.

Cet identifiant permet de référencer l'élément dans le code-behind, facilitant ainsi les interactions et la manipulation de l'élément à l'exécution.

```
<StackPanel Name="buttonContainer">  
    <Button Click="RemoveThis_Click">Click to remove this button</Button>  
</StackPanel>
```

```
private void RemoveThis_Click(object sender, RoutedEventArgs e)  
{  
    var element = (FrameworkElement)e.Source;  
  
    if (buttonContainer.Children.Contains(element))  
        buttonContainer.Children.Remove(element);  
}
```

# Les points clés des named element

- `x:Name` : Utilisé pour donner un nom unique à un élément.
- **Référence dans le code** : Le nom attribué permet de manipuler l'élément dans le code-behind.
- **Portée** : Les noms doivent être uniques dans la portée définie par l'élément racine de la page.

## Les propriétés attachées

Les propriétés attachées permettent de **définir des propriétés/valeur** supplémentaires sur **n'importe quel élément xml**.

Les propriétés attachées sont utilisées pour spécifier des valeurs uniques pour des propriétés définies dans un élément parent.

Elles sont généralement notées sous la forme `ownerType.propertyName`.

Le scénario le plus courant pour les propriétés attachées est de permettre aux éléments enfants de communiquer la valeur d'une propriété à leur élément parent.



## Exemple de propriété attachée

L'exemple suivant illustre la propriété attachée `DockPanel.Dock`.

1. La classe `DockPanel` parcourt ses éléments enfants.
2. Elle vérifie chaque élément pour une valeur définie de `DockPanel.Dock`.
3. Si une valeur est trouvée, elle est utilisée lors de la mise en page pour positionner les éléments enfants.

```
<DockPanel>  
  <Button DockPanel.Dock="Left" Width="100" Height="20">I am on the left</Button>  
  <Button DockPanel.Dock="Right" Width="100" Height="20">I am on the right</Button>  
</DockPanel>
```

## Événement attaché

Un **événement attaché** est un type d'événement défini dans une classe qui peut être utilisé par n'importe quel élément dans l'arborescence des éléments, même si cet élément ne définit pas ou n'hérite pas de l'événement.

Cela permet d'ajouter un gestionnaire d'événements à un élément arbitraire plutôt qu'à un élément qui définit ou possède l'événement.

Un scénario possible est l'événement `Mouse.MouseDown` qui peut être placé sur n'importe quel `UIElement`.

## Les types de base en xml

Certaines classes, notamment les classes abstraites comme `ButtonBase`, ne peuvent pas être directement mappées à des éléments xml mais sont importantes pour l'héritage.

- `FrameworkElement` est la classe de base concrète pour les éléments d'interface utilisateur dans WPF
- `FrameworkContentElement` est une classe de base similaire pour les éléments orientés document

Ensemble, ces classes fournissent des propriétés communes configurables sur la plupart des éléments xml concrets.

## Qu'est-ce qu'une **Dependency Property** ?

Les **Dependency Properties** sont un type spécial de propriétés, utilisées pour prendre en charge plusieurs fonctionnalités importantes de WPF, notamment le système de **liaison de données** (data binding), le système de **styles**, les **animations**, et la gestion des **valeurs héritées** ou par **défaut**.

# Comment fonctionne une Dependency Property ?

Lorsqu'une Dependency Property est créée, elle est enregistrée dans le système WPF via la méthode `DependencyProperty.Register`.

La plupart des types de WPF implémentent des `DependencyProperty` :

```
<Button Content="Cliquez ici" Width="{Binding LargeurBouton}" />
```

Dans cet exemple, `Width` est une Dependency Property qui utilise la liaison de données pour obtenir sa valeur.

Les Dependency Properties sont un élément clé de WPF, car elles rendent l'interface plus flexible et permettent de répondre aux besoins d'une interface utilisateur réactive.

# Binding

# Principes du DataBinding

Le DataBinding en WPF permet de lier des données à des éléments d'interface. Il relie la logique métier et l'interface utilisateur sans code supplémentaire.

```
<TextBlock Text="{Binding NomUtilisateur}" />
```

- Utilise `{Binding}` pour relier une propriété du code à un composant de l'interface.
- La source peut être un objet, une propriété statique, ou même une source de données comme une base SQL.

## OneWay

Le mode **OneWay** crée un binding unidirectionnel de la source de données vers l'interface utilisateur.

- Idéal pour afficher des données statiques ou rarement mises à jour.
- Les modifications de la source sont reflétées dans l'interface, mais l'inverse n'est pas vrai.

```
<TextBlock Text="{Binding NomUtilisateur, Mode=OneWay}" />
```

Ici, le `TextBlock` affiche la valeur de `NomUtilisateur`, mais les modifications sur `TextBlock` ne changent pas `NomUtilisateur`.



## TwoWay

Le mode **TwoWay** permet un binding bidirectionnel entre la source et l'interface.

- Toute modification dans le contrôle d'interface met à jour la source.
- Inversement, les changements de la source sont reflétés dans le contrôle.

```
<TextBox Text="{Binding NomUtilisateur, Mode=TwoWay}" />
```

Ici, toute modification dans le `TextBox` met à jour `NomUtilisateur`, et vice versa.

## OneTime

Le mode **OneTime** initialise la valeur du contrôle à partir de la source de données une seule fois.

- Utile pour des données qui ne changent jamais ou qui n'ont pas besoin d'être mises à jour en temps réel.

```
<TextBlock Text="{Binding NomUtilisateur, Mode=OneTime}" />
```

`TextBlock` affichera la valeur initiale de `NomUtilisateur` au chargement, mais toute modification ultérieure de `NomUtilisateur` ne sera pas reflétée.

## OneWayToSource

Le mode **OneWayToSource** crée un binding unidirectionnel de l'interface utilisateur vers la source de données.

- Souvent utilisé pour capturer des données depuis l'interface sans les afficher directement.
- Les modifications dans le contrôle d'interface mettent à jour la source, mais les modifications de la source ne se reflètent pas dans le contrôle.

```
<TextBox Text="{Binding NomUtilisateur, Mode=OneWayToSource}" />
```

Ici, toute modification dans `TextBox` met à jour `NomUtilisateur`, mais si `NomUtilisateur` change, `TextBox` ne sera pas mis à jour.

## Default (Mode par défaut)

En fonction du type de contrôle, WPF applique automatiquement un mode de binding par défaut.

- **OneWay** pour les contrôles d'affichage comme `TextBlock` ou `Label`.
- **TwoWay** pour les contrôles d'édition, comme `TextBox` ou `Slider`.

**Exemple** (sans spécifier de mode) :

```
<TextBox Text="{Binding NomUtilisateur}" />
```

Dans cet exemple, `TextBox` utilisera par défaut le mode `TwoWay` pour permettre la mise à jour de `NomUtilisateur` depuis l'interface et inversement.

# Binding entre composants graphiques

Il est possible de lier les valeurs entre différents contrôles d'interface.

## Exemple :

```
<Slider x:Name="SliderVolume" Minimum="0" Maximum="100" />
<ProgressBar Value="{Binding Value, ElementName=SliderVolume}" />
```

- Ici, la `ProgressBar` est liée au `Slider`.
- `ElementName=SliderVolume` permet de lier la `ProgressBar` à la valeur du `Slider`.

# Binding avec les objets métier

Le Binding en WPF peut utiliser des objets métier pour représenter des données.

Classe `Utilisateur` :

```
public class Utilisateur
{
    public string Nom { get; set; }
    public int Age { get; set; }
}
```

XML :

```
<TextBlock Text="{Binding Utilisateur.Nom}" />
<TextBlock Text="{Binding Utilisateur.Age}" />
```

- La source de données est un objet métier `Utilisateur`.
- Permet de relier les propriétés des objets aux éléments visuels.

# DataTemplates

Les DataTemplates permettent de définir des styles de présentation pour les objets de données.

```
<Window.Resources>
    <DataTemplate x:Key="TemplateUtilisateur">
        <StackPanel>
            <TextBlock Text="{Binding Nom}" />
            <TextBlock Text="{Binding Age}" />
        </StackPanel>
    </DataTemplate>
</Window.Resources>

<ListBox ItemsSource="{Binding ListeUtilisateurs}" ItemTemplate="{StaticResource TemplateUtilisateur}" />
```

- **DataTemplate** détermine comment chaque objet **Utilisateur** sera affiché.
- **ItemTemplate** utilise ce template pour afficher chaque élément de **ListeUtilisateurs**.

# INotifyPropertyChanged

L'interface **INotifyPropertyChanged** permet de notifier l'interface des changements dans les données.

- L'interface `INotifyPropertyChanged` déclenche un événement quand une propriété change.
- Nécessaire pour que le Binding en WPF détecte et reflète les mises à jour en temps réel.



# Exemple

```
public class Utilisateur : INotifyPropertyChanged
{
    private string nom;
    public string Nom
    {
        get { return nom; }
        set
        {
            if (nom != value)
            {
                nom = value;
                OnPropertyChanged("Nom");
            }
        }
    }
    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

## Conclusion

Le DataBinding en WPF est un puissant outil pour lier l'interface graphique et la logique métier. Avec des concepts comme Two Way Binding, les DataTemplates, et l'interface `INotifyPropertyChanged`, il est possible de créer des applications riches et interactives en séparant la logique de présentation et les données.

# Styles et positionnements

# Introduction au système de layout

- Le layout en WPF définit la manière dont les éléments sont disposés dans une interface.
- Contrôle de la taille, de la position, et de l'espacement des éléments.
- Composants essentiels : conteneurs, panneaux, et propriétés de layout.

## Les conteneurs de layout

- **Grid** : Disposition en lignes et colonnes, contrôle précis de la structure.
- **StackPanel** : Empile les éléments verticalement ou horizontalement.
- **DockPanel** : Positionne les éléments en fonction de leurs bords.
- **Canvas** : Disposition absolue avec coordonnées X et Y fixes.
- **WrapPanel** : Enroule les éléments lorsqu'ils débordent.

## Le conteneur Grid

- **Grid** permet de diviser l'interface en lignes et colonnes.
- Idéal pour des mises en page structurées et complexes.
- Chaque cellule peut contenir un ou plusieurs éléments.

# Création d'une grille (Grid) de base

- Utilisation des propriétés `RowDefinitions` et `ColumnDefinitions`.
- Exemple d'une grille 2x2 :

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="2*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
</Grid>
```

# Placement d'éléments dans un Grid

- Utilisation des propriétés `Grid.Row` et `Grid.Column` pour placer un élément.

- Exemple :

```
<Button Content="Bouton 1" Grid.Row="0" Grid.Column="0" />  
<Button Content="Bouton 2" Grid.Row="1" Grid.Column="1" />
```

- Les propriétés **RowSpan** et **ColumnSpan** permettent à un élément de s'étendre sur plusieurs lignes ou colonnes.



# Utilisation avancée de Grid

- Taille des lignes/colonnes :
  - **Auto** : S'adapte à la taille de son contenu.
  - **Fixed** : Taille fixe définie (e.g., 100 pixels).
  - **Star (\*)** : Utilise la place restante en fonction d'un ratio.

Exemple d'une ligne avec Auto, 2\*, et 100 pixels de hauteur :

```
<Grid.RowDefinitions>  
  <RowDefinition Height="Auto" />  
  <RowDefinition Height="2*" />  
  <RowDefinition Height="100" />  
</Grid.RowDefinitions>
```

## Le conteneur StackPanel

- `StackPanel` dispose les éléments en une pile verticale ou horizontale.
- Simple à utiliser pour des mises en page linéaires.

# Création d'un StackPanel vertical

`Orientation="Vertical"` empile les éléments verticalement (par défaut).

Exemple :

```
<StackPanel Orientation="Vertical">  
  <Button Content="Bouton 1" />  
  <Button Content="Bouton 2" />  
</StackPanel>
```

# Création d'un StackPanel horizontal

`Orientation="Horizontal"` aligne les éléments horizontalement.

Exemple :

```
<StackPanel Orientation="Horizontal">  
  <Button Content="Bouton 1" />  
  <Button Content="Bouton 2" />  
</StackPanel>
```

## Limitations du StackPanel

- Pas de contrôle direct de la disposition en plusieurs colonnes/lignes comme avec `Grid`.
- Prend automatiquement la taille des éléments, peut provoquer du débordement si l'espace est limité.

## Le conteneur DockPanel

- `DockPanel` permet de placer des éléments en fonction des bords (haut, bas, gauche, droite).
- Idéal pour des mises en page où des éléments sont ancrés aux bords de la fenêtre.

# Utilisation de DockPanel

- Propriété `DockPanel.Dock` pour spécifier l'ancrage de chaque élément.
- Exemple :

```
<DockPanel>  
  <Button Content="En haut" DockPanel.Dock="Top" />  
  <Button Content="À gauche" DockPanel.Dock="Left" />  
  <Button Content="Contenu principal" />  
</DockPanel>
```

# Propriété LastChildFill

- `LastChildFill="True"` permet au dernier élément de remplir tout l'espace restant.
- Exemple avec `LastChildFill` activé (par défaut) :

```
<DockPanel LastChildFill="True">  
  <Button Content="En haut" DockPanel.Dock="Top" />  
  <Button Content="À gauche" DockPanel.Dock="Left" />  
  <Button Content="Rempli l'espace restant" />  
</DockPanel>
```



## Le conteneur Canvas

- Canvas offre un positionnement absolu par coordonnées (X, Y).
- Idéal pour les interfaces nécessitant un contrôle exact du placement.

# Positionnement dans un Canvas

- Propriétés `Canvas.Left`, `Canvas.Top` pour définir les positions X et Y de chaque élément.
- Exemple :

```
<Canvas>  
  <Button Content="Positionné" Canvas.Left="50" Canvas.Top="100" />  
  <Button Content="Positionné" Canvas.Left="150" Canvas.Top="200" />  
</Canvas>
```

## Limites du Canvas

- Ne s'adapte pas automatiquement aux changements de taille d'écran.
- Utilisé principalement pour des interfaces fixes ou des dessins.

## Le conteneur WrapPanel

- `WrapPanel` dispose les éléments horizontalement ou verticalement et les enroule automatiquement.
- Pratique pour les éléments de taille dynamique ou pour les galeries d'images.

# Utilisation de WrapPanel en disposition horizontale

- `Orientation="Horizontal"` : Les éléments s'enroulent sur la ligne suivante lorsqu'ils atteignent le bord.
- Exemple :

```
<WrapPanel Orientation="Horizontal">  
  <Button Content="Bouton 1" Width="100" />  
  <Button Content="Bouton 2" Width="100" />  
  <Button Content="Bouton 3" Width="100" />  
</WrapPanel>
```

# Utilisation de WrapPanel en disposition verticale

- `Orientation="Vertical"` : Les éléments s'enroulent verticalement.
- Exemple :

```
<WrapPanel Orientation="Vertical">  
  <Button Content="Bouton 1" Height="50" />  
  <Button Content="Bouton 2" Height="50" />  
</WrapPanel>
```

## Limites du WrapPanel

- Le dimensionnement des éléments doit être contrôlé pour éviter des mises en page désorganisées.
- Moins adapté pour des dispositions strictement organisées que le `Grid`.

# Comparaison des Conteneurs de Layout

Conteneur	Utilisation principale	Orientation
<b>Grid</b>	Mise en page structurée en lignes/colonnes	N/A
<b>StackPanel</b>	Disposition linéaire simple	Verticale ou Horizontale
<b>DockPanel</b>	Ancrage aux bords	Basé sur le dock
<b>Canvas</b>	Positionnement absolu	N/A
<b>WrapPanel</b>	Disposition enroulée	Verticale ou Horizontale



# Propriétés de Positionnement

- **Margin** : Contrôle l'espace extérieur autour d'un élément.
- **Padding** : Contrôle l'espace intérieur entre les bords et le contenu.
- **HorizontalAlignment** et **VerticalAlignment** : Alignement des éléments (Gauche, Droite, Centre, Étirement).
- **ZIndex** : Gère l'ordre d'empilement pour superposer les éléments.

# Utilisation des grilles et des colonnes

- Configuration des lignes et colonnes dans un `Grid` pour un layout structuré.
- Propriétés **RowSpan** et **ColumnSpan** pour étendre un élément sur plusieurs lignes/colonnes.
- Fractionnement avec des tailles **auto**, **étoile**, et **fixe** pour des mises en page dynamiques.

# Introduction aux Styles en WPF

- Les **styles** en WPF permettent de définir et centraliser l'apparence des contrôles.
- Utilisés pour homogénéiser l'apparence sans répéter les propriétés de style dans chaque élément.
- Un style peut être appliqué à un contrôle unique, un type de contrôle spécifique, ou globalement.

# Déclaration de Styles en XAML

- Les styles sont déclarés avec la balise `<Style>` en XAML.
- Utilisation de la propriété **TargetType** pour spécifier le type de contrôle visé.

```
<Style TargetType="Button">  
    <Setter Property="Background" Value="LightBlue" />  
    <Setter Property="FontSize" Value="14" />  
</Style>
```

- **Setter** : Définit une propriété et une valeur pour un style.

## Propriétés de Style

- **TargetType** : Spécifie le type de contrôle pour lequel le style est conçu (`Button`, `TextBox`, etc.).
- **Setters** : Définissent les propriétés de style (couleur, taille, police, etc.).
- **BasedOn** : Permet d'hériter d'un autre style pour réutiliser des propriétés existantes.

## Exemple de style basé sur un autre style

```
<Style x:Key="BaseButtonStyle" TargetType="Button">  
    <Setter Property="Background" Value="LightGray" />  
</Style>  
  
<Style TargetType="Button" BasedOn="{StaticResource BaseButtonStyle}">  
    <Setter Property="FontSize" Value="16" />  
</Style>
```

# Application de style via le XAML

- Les styles peuvent être appliqués de plusieurs manières :
  - **Globale** : S'applique automatiquement à tous les contrôles du type spécifié.
  - **Par clé (x:Key)** : Nécessite une référence explicite pour être appliqué à un contrôle.

# Style Global

- S'applique automatiquement à tous les contrôles du type défini sans `x:Key`.

```
<Style TargetType="Button">  
  <Setter Property="Background" Value="LightBlue" />  
</Style>
```



# Style avec Clé

- Nécessite de spécifier `x:Key` pour être appliqué manuellement.

```
<Style x:Key="CustomButtonStyle" TargetType="Button">  
    <Setter Property="Background" Value="LightGreen" />  
</Style>  
  
<Button Style="{StaticResource CustomButtonStyle}" Content="Bouton personnalisé" />
```

## Application d'un Style en C#

- Il est également possible d'appliquer des styles directement dans le code-behind en C#.

```
Button myButton = new Button();  
myButton.Style = (Style)FindResource("CustomButtonStyle");
```

# Styles Implicites

- Les **styles implicites** sont appliqués à tous les contrôles d'un type sans nécessiter `x:Key`.
- Permettent une uniformité automatique pour tous les contrôles de même type.

```
<Style TargetType="TextBox">  
  <Setter Property="FontSize" Value="14" />  
</Style>
```

- Tous les `TextBox` de l'application adopteront ce style.

## Styles Explicites

Nécessitent une clé (`x:Key`) et doivent être appliqués explicitement.

Exemple :

```
<Style x:Key="LargeButtonStyle" TargetType="Button">  
    <Setter Property="FontSize" Value="18" />  
</Style>
```

Utilisation explicite :

```
<Button Content="Grand Bouton" Style="{StaticResource LargeButtonStyle}" />
```

# Introduction aux Ressources

- Les **ressources** permettent de stocker des valeurs réutilisables comme les couleurs, tailles, styles.
- WPF permet de déclarer des ressources au niveau local, de la fenêtre, ou globales.

## Ressources locales

- Déclarées directement dans un contrôle, elles ne sont accessibles qu'à ce contrôle.

```
<Button>  
  <Button.Resources>  
    <Color x:Key="ButtonBackground">LightCoral</Color>  
  </Button.Resources>  
</Button>
```

# Ressources globales

- Déclarées au niveau de l'application ou de la fenêtre, elles peuvent être partagées.

```
<Window.Resources>  
  <Color x:Key="PrimaryColor">CornflowerBlue</Color>  
</Window.Resources>
```

Utilisation :

```
<Button Background="{StaticResource PrimaryColor}" />
```

# Utilisation des Dictionnaires de Ressources

- Permettent de regrouper et de gérer des styles dans des fichiers séparés.
- Utilisation courante pour les **thèmes** et les **styles réutilisables**.

Exemple d'inclusion d'un dictionnaire de ressources :

```
<ResourceDictionary Source="Styles/ControlStyles.xaml" />
```



# Exemple de fichier de dictionnaire de ressources

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Style TargetType="Button">
    <Setter Property="Background" Value="LightSteelBlue" />
    <Setter Property="FontSize" Value="16" />
  </Style>
</ResourceDictionary>
```

# Triggers dans les Styles

- Les **triggers** permettent de changer les propriétés en fonction d'événements ou d'états spécifiques.
- Triggers courants : **Property Triggers**, **Data Triggers**, et **Event Triggers**.

# Property Trigger

Change les propriétés d'un contrôle lorsque la valeur d'une propriété change.

```
<Style TargetType="Button">
  <Setter Property="Background" Value="LightBlue"/>
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Background" Value="Orange"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

## Gestion de Thèmes avec les Styles

- WPF supporte les thèmes : les styles peuvent être chargés selon le thème du système.
- Différentes **ressources** peuvent être appliquées pour des thèmes sombres/clairs.

# Utilisation de MergedDictionaries pour les Thèmes

Les MergedDictionaries peuvent charger des ressources conditionnelles pour changer de thème.

Exemple :

```
<ResourceDictionary>  
  <ResourceDictionary.MergedDictionaries>  
    <ResourceDictionary Source="Themes/LightTheme.xaml" />  
  </ResourceDictionary.MergedDictionaries>  
</ResourceDictionary>
```


## Qu'est-ce qu'un Behavior ?

Un Behavior en WPF est une manière de définir et de réutiliser une logique spécifique dans des contrôles UI.

- Ajoute un comportement sans modifier directement la classe du contrôle.
- Favorise la séparation des préoccupations et la réutilisabilité.

Exemple : Vous avez des interactions complexes ou non standard (drag-and-drop, double-clic, etc.)

# Les types de Behaviors

1. *Attached Behaviors* : Définis via des propriétés attachées.
2. *Blend Behaviors* : Utilisent des classes prêtes à l'emploi (nécessitent `System.Windows.Interactivity` ou encore `Microsoft.Xaml.Behaviors.Wpf` ).

## Définir un Attached Behavior

Les comportements attachés utilisent des **propriétés attachées** pour ajouter du comportement. Cela permet de manipuler ou d'écouter des événements d'un contrôle.

La structure d'un Attached Behavior est la suivante:

- Une propriété attachée (avec getter et setter).
- Gestion d'événements ou logique dans le setter.
- Nettoyage des ressources si nécessaire.



# Exemple d'attached property

```
public static class FocusBehavior
{
    public static readonly DependencyProperty IsFocusedProperty =
        DependencyProperty.RegisterAttached(
            "IsFocused",
            typeof(bool),
            typeof(FocusBehavior),
            new PropertyMetadata(false, OnIsFocusedChanged));

    public static bool GetIsFocused(DependencyObject obj) =>
        (bool)obj.GetValue(IsFocusedProperty);

    public static void SetIsFocused(DependencyObject obj, bool value) =>
        obj.SetValue(IsFocusedProperty, value);

    private static void OnIsFocusedChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
    {
        if (d is UIElement element && e.NewValue is bool isFocused && isFocused)
        {
            element.Focus();
        }
    }
}
```

```
<TextBox local:FocusBehavior.IsFocused="True" />
```

## Définir un Blend Behavior

Un Blend Behavior dérive de `Behavior<T>` où `T` est le type de contrôle ciblé. Pour hériter de cette classe il faut l'importer depuis l'un des packages cité précédemment.

Pour créer un Blend Behavior il faut:

1. Hériter de `Behavior<T>`.
2. Surcharger les méthodes `OnAttached` et `OnDetaching`.
3. Gérer l'interaction avec le contrôle parent.

# Exemple de Blend Behavior

```
using System.Windows;
using System.Windows.Interactivity;

public class DragBehavior : Behavior<UIElement>
{
    protected override void OnAttached()
    {
        AssociatedObject.MouseLeftButtonDown += OnMouseLeftButtonDown;
    }

    protected override void OnDetaching()
    {
        AssociatedObject.MouseLeftButtonDown -= OnMouseLeftButtonDown;
    }

    private void OnMouseLeftButtonDown(object sender, MouseButtonEventArgs e)
    {
        MessageBox.Show("Element dragged!");
    }
}
```

```
<Rectangle Width="100" Height="100" Fill="Blue">
    <i:Interaction.Behaviors>
        <local:DragBehavior />
    </i:Interaction.Behaviors>
</Rectangle>
```

## Qu'est-ce qu'une animation en WPF ?

Les animations en WPF permettent de modifier des propriétés visuelles d'un élément au fil du temps. Cela peut concerner la position, la taille, la couleur, ou d'autres aspects graphiques.

### **Exemples d'animations :**

- Déplacer un bouton sur l'écran.
- Modifier la couleur d'arrière-plan d'une fenêtre.
- Faire grandir ou rétrécir une image.

## Types d'animations en WPF

WPF propose plusieurs types d'animations, dont les plus courants sont :

- **DoubleAnimation** : anime un nombre à virgule flottante (par exemple, la position ou l'opacité).
- **ColorAnimation** : anime les couleurs.
- **Storyboard** : une collection d'animations qui peuvent être lancées ensemble.

## Exemple de DoubleAnimation

Voici un exemple d'animation de la propriété `Opacity` d'un bouton :

```
<Button Name="MyButton" Content="Click Me">
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.Click">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation Storyboard.TargetName="MyButton"
                           Storyboard.TargetProperty="Opacity"
                           From="1" To="0" Duration="0:0:2" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

# Exemple de ColorAnimation

Animation de la couleur de fond d'un élément :

```
<Rectangle Width="200" Height="100">
  <Rectangle.Fill>
    <SolidColorBrush x:Name="FillBrush" Color="Blue"/>
  </Rectangle.Fill>
  <Rectangle.Triggers>
    <EventTrigger RoutedEvent="Rectangle.MouseEnter">
      <BeginStoryboard>
        <Storyboard>
          <ColorAnimation Storyboard.TargetName="FillBrush"
            Storyboard.TargetProperty="Color"
            From="Blue" To="Red" Duration="0:0:1"/>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Rectangle.Triggers>
</Rectangle>
```

# Storyboard : Unification des Animations

Un **Storyboard** permet de regrouper plusieurs animations et de les exécuter en même temps ou séquentiellement. Par exemple, une animation de déplacement et de changement de couleur en même temps.

```
<Button Name="AnimateButton" Content="Animate">
  <Button.Resources>
    <Storyboard x:Key="buttonAnimation">
      <DoubleAnimation Storyboard.TargetName="AnimateButton"
        Storyboard.TargetProperty="(Button.Width)"
        From="100" To="200" Duration="0:0:1" />
      <ColorAnimation Storyboard.TargetName="AnimateButton"
        Storyboard.TargetProperty="(Button.Background).(SolidColorBrush.Color)"
        From="Green" To="Yellow" Duration="0:0:1" />
    </Storyboard>
  </Button.Resources>
  <Button.Triggers>
```

```
<EventTrigger RoutedEvent="Button.Click">
  <BeginStoryboard Storyboard="{StaticResource buttonAnimation}" />
</EventTrigger>
</Button.Triggers>
</Button>
```



# Utilisation des Événements avec les Animations

Les animations peuvent être déclenchées par des événements comme un clic de bouton, le survol de la souris ou un changement de valeur.

## Exemples d'événements :

- `MouseEnter`
- `Click`
- `Loaded`

# Contrôler les Animations avec le Code C#

Les animations peuvent aussi être manipulées directement depuis le code C#.

Exemple pour démarrer une animation dans C# :

```
DoubleAnimation animation = new DoubleAnimation
{
    From = 0,
    To = 100,
    Duration = TimeSpan.FromSeconds(1)
};
MyElement.BeginAnimation(UIElement.WidthProperty, animation);
```

## Qu'est-ce que le MVVM ?

Le pattern **MVVM** (Model-View-ViewModel) est un modèle d'architecture permettant une séparation claire des responsabilités.

- **Model** : Représente les données et la logique métier de l'application.
- **View** : Interface utilisateur, responsable de l'affichage des données.
- **ViewModel** : Sert d'intermédiaire entre le **Model** et la **View**, contenant la logique de présentation et les données préparées pour l'affichage.

## Pourquoi utiliser le MVVM ?

- **Séparation des préoccupations** : Chaque composant a une responsabilité bien définie.
- **Testabilité** : La logique est séparée de l'interface, facilitant les tests unitaires.
- **Maintenance facilitée** : Permet de modifier l'interface ou la logique métier indépendamment.
- **Réutilisation** : Le **ViewModel** peut être réutilisé dans différentes interfaces utilisateur (WPF, Xamarin, etc.).

## Rôle du Model

Le **Model** représente les données brutes de l'application et la logique métier associée.

- Gère les données de l'application (Utilisation du design pattern Repository).
- Contient la logique métier (par exemple, des calculs ou des validations).
- Ne doit pas avoir de dépendance directe à la **View** ou au **ViewModel**.

## Rôle de la View

La **View** est responsable de l'affichage des données et de la gestion des interactions avec l'utilisateur.

- Affiche les informations fournies par le **ViewModel** via des liaisons de données (data-binding).
- Ne contient pas de logique métier, seulement des éléments d'interface.
- Peut inclure des événements utilisateur, mais les actions sont généralement gérées par des commandes dans le **ViewModel**.

## Rôle du **ViewModel**

Le **ViewModel** est l'abstraction de la logique de la vue, contenant la logique de présentation. Il prépare les données du **Model** pour les afficher dans la **View** et gère la logique des commandes (actions déclenchées par l'utilisateur).

- Expose des propriétés qui seront liées à la **View**.
- Expose des commandes que la **View** peut exécuter en réponse aux actions de l'utilisateur.
- Manipule le **Model** et le prépare pour l'affichage dans la **View**.

## Communication entre les composants

- **View** et **ViewModel** : La **View** se lie aux propriétés et aux commandes du **ViewModel** via le data-binding (liaison de données).
- **ViewModel** et **Model** : Le **ViewModel** interagit avec le **Model** pour obtenir ou manipuler les données nécessaires à l'affichage.



# Mise en place du MVVM dans une application WPF

Voici les étapes pour mettre en place le MVVM dans une application WPF en suivant les bonnes pratiques.

## Étape 1 : Créer le Model

Le **Model** représente les données et la logique métier de l'application. Il doit être indépendant de la **View** et du **ViewModel**.

Exemple de **Model** dans WPF :

```
// Modèle de données représentant un utilisateur
public class UserModel
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

## Étape 2 : Créer le ViewModel

Le **ViewModel** expose les propriétés que la **View** va lier. Il contient également la logique nécessaire à la gestion des commandes. Le **ViewModel** ne doit pas contenir de logique d'affichage, mais seulement la logique métier nécessaire à la présentation des données.

# Exemple de ViewModel

```
public class UserViewModel : INotifyPropertyChanged
{
    private UserModel _user;
    public string Name => _user.Name;
    public int Age => _user.Age;

    public UserViewModel()
    {
        _user = new UserModel { Name = "Alice", Age = 30 };
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```

## Étape 3 : Créer la View (XAML)

La **View** est construite en XAML, où elle se lie aux propriétés et aux commandes exposées par le **ViewModel**.

```
<Window x:Class="MVVMApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MVVM Example" Height="200" Width="300">
    <Grid>
        <TextBox Text="{Binding Name}" Width="200" Height="30" VerticalAlignment="Top" Margin="10"/>
        <TextBlock Text="{Binding Age}" VerticalAlignment="Top" Margin="10,50,10,10"/>
    </Grid>
</Window>
```

Dans ce cas, le **ViewModel** est automatiquement connecté à la **View** grâce à la liaison de données.

## Étape 4 : Lier le ViewModel à la View

Une fois que le **ViewModel** est créé, il doit être associé à la **View** via la propriété `DataContext`. Cela peut se faire dans le code-behind ou en XAML.

Exemple dans le code-behind de la **View** :

```
public MainWindow()  
{  
    InitializeComponent();  
    DataContext = new UserViewModel(); // Lier le ViewModel à la View  
}
```

Cela permet à la **View** d'accéder aux propriétés du **ViewModel** et de les afficher via le data-binding.

## Étape 5 : Créer les Commandes

Les **Commandes** permettent de gérer les actions de l'utilisateur dans le MVVM sans avoir besoin de code-behind dans la **View**.

Les commandes sont généralement définies dans le **ViewModel** et liées à des éléments de l'interface utilisateur dans la **View**.

Exemple d'implémentation d'une commande dans WPF :

## Étape 5 : Créer les Commandes

```
public class RelayCommand : ICommand
{
    private readonly Action _execute;
    private readonly Func<bool> _canExecute;

    public RelayCommand(Action execute, Func<bool> canExecute = null)
    {
        _execute = execute;
        _canExecute = canExecute ?? (() => true);
    }

    public bool CanExecute(object parameter) => _canExecute();

    public void Execute(object parameter) => _execute();

    public event EventHandler CanExecuteChanged
    {
        add => CommandManager.RequerySuggested += value;
        remove => CommandManager.RequerySuggested -= value;
    }
}
```



## Étape 5 : Créer les Commandes

Dans le **ViewModel**, vous définissez la commande :

```
public ICommand SubmitCommand { get; }

public UserViewModel()
{
    SubmitCommand = new RelayCommand(OnSubmit, CanSubmit);
}

private void OnSubmit()
{
    // Logique de soumission
}

private bool CanSubmit()
{
    return !string.IsNullOrEmpty(Name);
}
```

## Étape 6 : Lier la Commande à la View

La **View** liera la commande à un bouton ou un autre élément interactif. Exemple en XAML :

```
<Button Content="Submit" Command="{Binding SubmitCommand}" />
```

Cela permet d'exécuter la commande **SubmitCommand** du **ViewModel** lorsque l'utilisateur clique sur le bouton.

## Bonnes pratiques du MVVM

1. **Séparer la logique** : Le **Model** ne doit contenir que la logique métier et les données. La **View** ne doit contenir aucune logique de traitement, seulement l'affichage.
2. **Data-binding** : Utiliser le data-binding pour lier les propriétés et les commandes entre la **View** et le **ViewModel**.

## Bonnes pratiques du MVVM

3. **Utiliser des commandes** : Toujours utiliser des commandes pour les interactions de l'utilisateur (clics de boutons, etc.), au lieu de gérer des événements dans le code-behind.
4. **INotifyPropertyChanged** : Implémenter l'interface `INotifyPropertyChanged` dans le **ViewModel** pour notifier la **View** des changements de données.

## Contrôles de base

- **Button** : Un bouton pour les actions utilisateur.
- **TextBox** : Zone de texte permettant à l'utilisateur de saisir du texte.
- **Label** : Affiche du texte statique.

## Contrôles de sélection

- **ComboBox** : Liste déroulante permettant de choisir une option parmi plusieurs.
- **ListBox** : Liste permettant de sélectionner un ou plusieurs éléments.
- **CheckBox** : Case à cocher pour une option binaire.
- **RadioButton** : Boutons radio pour choisir une seule option dans un groupe.

## Contrôles de mise en forme

- **TextBlock** : Affiche du texte simple ou formaté.
- **RichTextBox** : Permet d'afficher et de modifier du texte avec mise en forme.
- **PasswordBox** : Zone de saisie de texte masqué pour les mots de passe.

## Contrôles de contenu

- **ContentControl** : Conteneur pour un seul élément enfant.
- **ContentPresenter** : Affiche le contenu d'un contrôle, comme pour un modèle de contrôle.
- **Image** : Affiche des images dans l'interface.



## Contrôles multimédia

- **MediaElement** : Permet d'intégrer des fichiers multimédia (audio, vidéo).
- **MediaPlayer** : Permet de jouer des fichiers multimédia en arrière-plan.

## Contrôles interactifs

- **Slider** : Permet de sélectionner une valeur dans une plage.
- **ProgressBar** : Affiche la progression d'une tâche.
- **SpinBox** : Permet d'ajuster une valeur numériquement avec des boutons de flèche.

## Contrôles de navigation

- **TabControl** : Permet de créer des onglets pour naviguer entre plusieurs vues.
- **Frame** : Conteneur permettant d'afficher différentes pages.
- **Hyperlink** : Permet de créer des liens interactifs dans une interface.

## Contrôles spécialisés

- **DataGrid** : Affiche des données sous forme de tableau.
- **TreeView** : Affiche des données sous forme d'arbre hiérarchique.
- **ListView** : Liste améliorée avec des fonctionnalités de tri et de mise en forme avancées.

## Contrôles de validation

- **Validation** : Permet de valider l'entrée des utilisateurs, avec gestion des erreurs de saisie.
- **ErrorTemplate** : Utilisé pour personnaliser l'affichage des erreurs de validation.

## Contrôles de défilement

- **ScrollView** : Conteneur permettant de faire défiler son contenu lorsque celui-ci dépasse la taille de la fenêtre.

# Composants graphiques

# Qu'est-ce qu'un **ControlTemplate** en WPF ?

- **Definition** : Un **ControlTemplate** permet de redéfinir complètement l'apparence visuelle d'un contrôle tout en conservant sa logique fonctionnelle.
- **Exemple** : Changer l'apparence d'un bouton.

```
<Button Content="Cliquez-moi">
  <Button.Template>
    <ControlTemplate TargetType="Button">
      <Grid>
        <Ellipse Fill="LightBlue" />
        <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center"/>
      </Grid>
    </ControlTemplate>
  </Button.Template>
</Button>
```



## Qu'est-ce qu'un **ItemsControl** en WPF ?

- **Definition** : Un **ItemsControl** est un conteneur générique pour afficher une liste d'éléments avec personnalisation possible de leur présentation.

```
<ItemsControl>
  <ItemsControl.Items>
    <TextBlock Text="Alice" />
    <TextBlock Text="Bob" />
    <TextBlock Text="Charlie" />
  </ItemsControl.Items>
</ItemsControl>
```

- **Remarque** : Peut être lié à une source de données avec `ItemsSource`

.

## Qu'est-ce que la Conversion en WPF ?

- **Definition** : Un **convertisseur** (via `IValueConverter`) permet de transformer une valeur de donnée avant son affichage.
- **Exemple** : Convertir une valeur booléenne en couleur.

# Exemple de convertisseur

## C# : Convertisseur

```
public class BoolToColorConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (bool)value ? "Green" : "Red";
    }
    public object ConvertBack(...) => throw new NotImplementedException();
}
```

## XAML : Utilisation

```
<Window.Resources>
    <local:BoolToColorConverter x:Key="BoolToColor" />
</Window.Resources>
<Rectangle Width="100" Height="50" Fill="{Binding IsActive, Converter={StaticResource BoolToColor}}" />
```

## Qu'est-ce que la Validation en WPF ?

- **Definition** : La validation en WPF garantit que les données entrées par l'utilisateur respectent certaines règles.
- **Exemple** : Validation d'un champ avec des règles personnalisées.

### C# : ValidationRule

```
public class NotEmptyValidation : ValidationRule
{
    public override ValidationResult Validate(object value, CultureInfo cultureInfo)
    {
        return string.IsNullOrEmpty(value as string)
            ? new ValidationResult(false, "Le champ ne peut pas être vide.")
            : ValidationResult.ValidResult;
    }
}
```

# Validation dans le XAML

```
<TextBox>
    <TextBox.Text>
        <Binding Path="Nom" UpdateSourceTrigger="PropertyChanged">
            <Binding.ValidationRules>
                <local:NotEmptyValidation />
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

# Principaux patterns et idiomes

# Command dans WPF

- **Command** : Un mécanisme pour séparer la logique métier de l'interface utilisateur.
- Utilisé pour les actions déclenchées par des contrôles (boutons, menus).

```
public class MyCommand : ICommand
{
    public event EventHandler CanExecuteChanged;
    public bool CanExecute(object parameter) => true;
    public void Execute(object parameter) => MessageBox.Show("Command executed!");
}
```

# Usage d'une commande dans la vue

- Utilisation :

```
<Button Command="{Binding MyCommand}" Content="Click Me" />
```



# RelayCommand

- **RelayCommand** : Implémentation simplifiée de `ICommand`.
- Permet de définir la logique directement dans le ViewModel via des délégués.

# Exemple RelayCommand

```
public class RelayCommand : ICommand
{
    private readonly Action<object> _execute;
    private readonly Predicate<object> _canExecute;

    public RelayCommand(Action<object> execute, Predicate<object> canExecute = null)
    {
        _execute = execute;
        _canExecute = canExecute;
    }

    public bool CanExecute(object parameter) => _canExecute?.Invoke(parameter) ?? true;
    public void Execute(object parameter) => _execute(parameter);
    public event EventHandler CanExecuteChanged;
}
```

# Utilisation dans le ViewModel

```
public RelayCommand MyRelayCommand { get; }  
public ViewModel()  
{  
    MyRelayCommand = new RelayCommand(p => MessageBox.Show("RelayCommand!"));  
}
```

# Introduction à EventToCommand

En WPF, les commandes sont souvent utilisées pour relier des interactions utilisateur (boutons, menus, etc.) à la logique métier dans ViewModel. Mais certains événements (comme `MouseEnter`, `SelectionChanged`) ne supportent pas nativement les commandes.

# Concept de EventToCommandBehavior

- **Définition :**

`EventToCommandBehavior` est un comportement attaché (`Attached Behavior`) qui permet d'exécuter une commande lorsqu'un événement WPF est déclenché.

- **Fonctionnalités principales :**

- Associer un événement WPF à une commande du ViewModel.
- Passer des arguments personnalisés à la commande.

## Code C# EventToCommandBehavior

```
public partial class MainViewModel : ObservableObject
{
    [RelayCommand]
    private void OnSelectionChanged(object selectedItem)
    {
        Debug.WriteLine($"Item sélectionné : {selectedItem}");
    }
}
```

# Code XAML EventToCommandBehavior

```
<StackPanel>
    <ComboBox Width="200" Margin="10"
        ItemsSource="{Binding Items}">
        <mvvm:Interaction.Behaviors>
            <mvvm:EventToCommandBehavior
                EventName="SelectionChanged"
                Command="{Binding OnSelectionChangedCommand}"
                CommandParameter="{Binding SelectedItem, RelativeSource={RelativeSource Self}}" />
        </mvvm:Interaction.Behaviors>
    </ComboBox>
</StackPanel>
```

# Détails du XAML

- `EventName="SelectionChanged"`
  - Définit l'événement WPF à écouter.
- `Command="{Binding OnSelectionChangedCommand}"`
  - Lie la commande dans le ViewModel.
- `CommandParameter="{Binding SelectedItem ...}"}"`
  - Passe l'élément sélectionné comme paramètre à la commande.



# Gestion des messages d'erreur

- **Messages d'erreur :**

1. Afficher des erreurs liées à la validation de l'utilisateur.
2. Utiliser **IDataErrorInfo** ou **INotifyDataErrorInfo**.

- **Exemple avec IDataErrorInfo :**

```
public string this[string columnName] =>
    columnName == "Name" && string.IsNullOrEmpty(Name)
    ? "Name cannot be empty."
    : null;

public string Error => null;
```

- Dans l'interface :

```
<TextBox Text="{Binding Name, ValidatesOnDataErrors=True}" />
```

# Internationalisation (i18n) en WPF

- **Objectif** : Gérer plusieurs langues dans l'application.
- **Méthodes courantes** :
  1. Utiliser des fichiers **.resx**.
  2. Associer des ressources à des éléments via `x:Static`.
- **Fichier de ressources** : `Strings.resx`  
Clé : `Hello` | Valeur : `Bonjour`  
Clé : `Goodbye` | Valeur : `Au revoir`

- **Utilisation dans XAML :**

```
<TextBlock Text="{x:Static resx:Strings.Hello}" />
```

- **Changement dynamique (via `CultureInfo`) :**

```
Thread.CurrentThread.CurrentUICulture = new CultureInfo("fr-FR");
```

