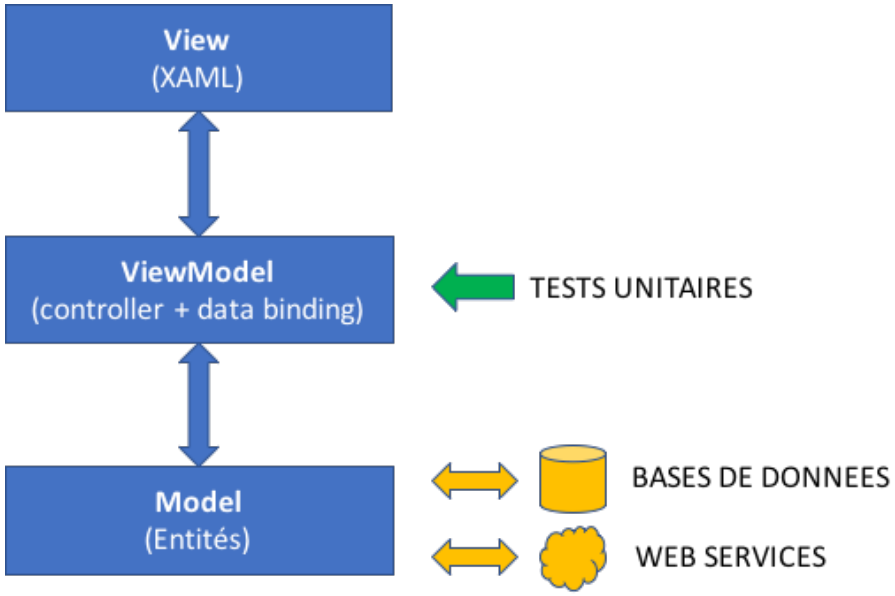
MVVM signifie *Model-View-ViewModel* :

- *Model* correspond aux données. Il s'agit en général de plusieurs classes qui permettent d'accéder aux données, comme une classe *Client*, une classe *Commande*, etc. Peu importe la façon dont on remplit ces données (base de données, service web,…), c'est ce modèle qui est manipulé pour accéder aux données.
- *View* correspond à tout ce qui sera affiché, comme la page ou la fenêtre, les boutons, etc. En pratique, il s'agit du fichier `.xaml`.
- *ViewModel*, que l'on peut traduire en « modèle de vue », constitue la colle entre le modèle et la vue. Il s'agit d'une classe qui fournit une abstraction de la vue. Ce modèle de vue s'appuie sur la puissance du binding pour mettre à disposition de la vue les données du modèle. Il s'occupe également de gérer les commandes (actions événementielles) que nous verrons un peu plus loin.



Le but de MVVM est de faire en sorte que la vue n'effectue aucun traitement : elle ne doit faire qu'afficher les données présentées par le ViewModel. C'est le ViewModel qui est chargé de réaliser les traitements et d'accéder au modèle. Les tests en seront facilités, car la vue ne contenant aucun code, aucun test d'IHM ne sera nécessaire. Seuls des tests unitaires sur le ViewModel pourront être codés.

Afin de simplifier l'application du pattern MVVM, nous allons utiliser le framework « CommunityToolkit.Mvvm ». Ce framework va ainsi nous aider à mettre en place ce pattern.

Remarques : il existe d'autres frameworks permettant d'appliquer le pattern MVVM tels que Prism, Okra, Caliburn.Micro, etc. On peut aussi coder le MVVM à la main (sans framework !).

1. Application du patron MVVM à l'application « Calculatrice »

1.1. Création du projet

Partir de la solution du TD6 – Partie 1.

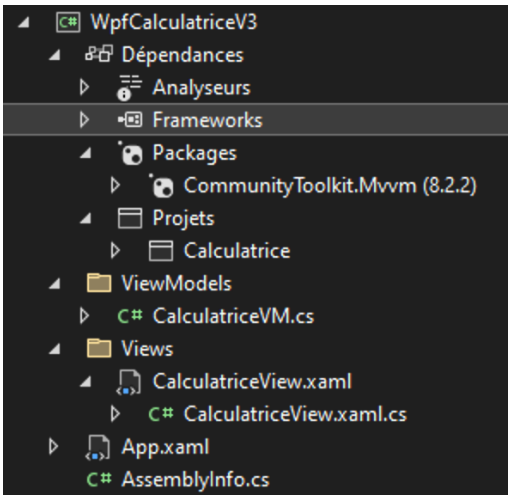
Ajouter un nouveau projet `WpfCalculatriceMVVM` de type « Application WPF » à la solution.

Ajouter le package NuGet « CommunityToolkit.Mvvm » (dernière version) au projet (Menu *Outils* > *Gestionnaire de packages NuGet* > *Gérer les packages NuGet pour la solution*).

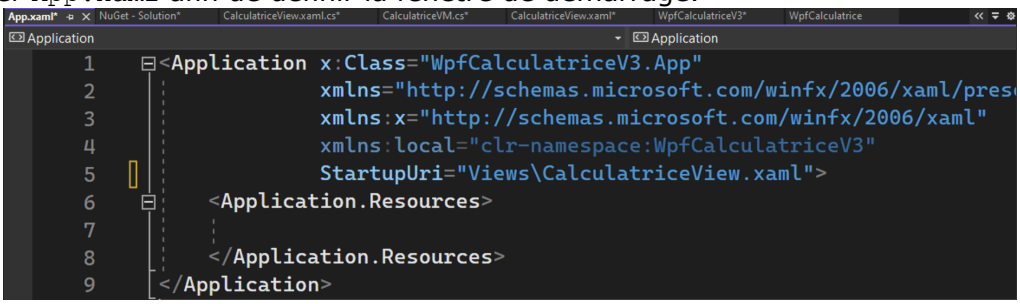
Supprimer le fichier `MainWindow.xaml`.

Créer un dossier `Views`. Ajouter une fenêtre WPF dans ce dossier.

Créer un dossier `ViewModels`. Ajouter la classe `CalculatriceVM` dans ce dossier.



Modifier la fichier `App.xaml` afin de définir la fenêtre de démarrage.



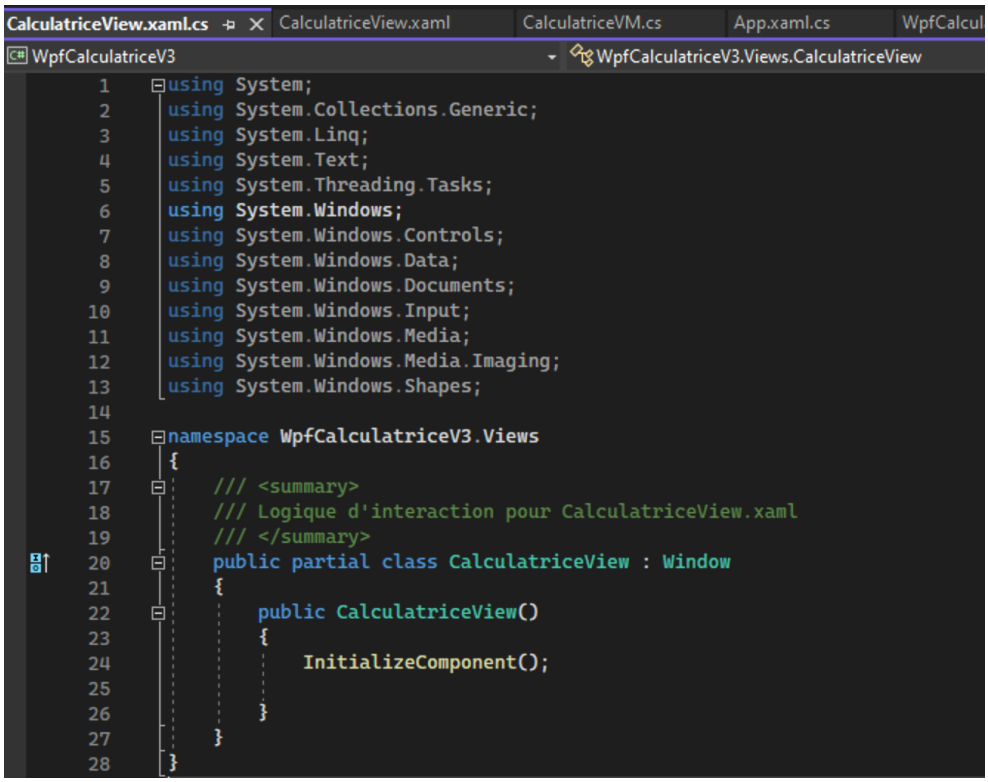
```
<Application x:Class="WpfCalculatriceV3.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WpfCalculatriceV3"
             StartupUri="Views\CalculatriceView.xaml">
    <Application.Resources>

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

Définir ce nouveau projet comme projet de démarrage de l'application. Lancer l'application pour voir si elle fonctionne.

1.2. Couche View

Le contenu XAML est le même que celui de l'autre fenêtre WPF. Le récupérer et supprimer les événements Click="..."

Code `cs` de la vue :



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace WpfCalculatriceV3.Views
{
    /// <summary>
    /// Logique d'interaction pour CalculatriceView.xaml
    /// </summary>
    public partial class CalculatriceView : Window
    {
        public CalculatriceView()
        {
            InitializeComponent();

        }
    }
}
```

Nous n'ajouterons qu'une seule ligne de code dans la vue. Tout le reste se fera dans le ViewModel.

Exécuter l'application pour voir si tout fonctionne bien.

1.3. Création du ViewModel

La classe `CalculatriceVM` est le ViewModel correspondant à la vue `CalculatriceView`. C'est cette classe qui coordonnera les échanges entre la vue et le modèle. Elle doit hériter de la classe de base `CommunityToolkit.Mvvm.ComponentModel.ObservableObject` fournie par MVVM Toolkit.

```
namespace WpfCalculatriceV3.ViewModels
{
    public class CalculatriceVM : ObservableObject
    {
```

Nous n'utilisons plus l'interface `INotifyPropertyChanged` comme dans les TDs précédents, mais `ObservableObject`, classe qui implémente l'interface `INotifyPropertyChanged` dans le cadre d'une architecture MVVM :
https://learn.microsoft.com/fr-fr/dotnet/communitytoolkit/mvvm/observableobject

1.4. Lien View - ViewModel

Pour le moment, la vue ne sait pas qu'un ViewModel lui est associé. Pour s'en rendre compte, ajouter dans le constructeur de `CalculatriceVM` le code suivant et positionner un point d'arrêt sur la ligne.

```
10  namespace WpfCalculatriceV3.ViewModels
11  {
12      public class CalculatriceVM : ObservableObject
13      {
14          public CalculatriceVM()
15          {
16              MessageBox.Show("Je passe par dans le View Model");
17          }
```

Exécuter ensuite l'application.
A aucun moment, le code n'est exécuté.

Rajouter maintenant le code suivant dans le fichier `.cs` de la vue :

```
namespace WpfCalculatriceV3.Views
{
    /// <summary>
    /// Logique d'interaction pour CalculatriceView.xaml
    /// </summary>
    public partial class CalculatriceView : Window
    {
        public CalculatriceView()
        {
            InitializeComponent();
            CalculatriceVM calculatriceVM = new CalculatriceVM();
            DataContext = calculatriceVM;
        }
    }
```

Ce code permet de lier la vue à son ViewModel et de définir le DataContext qui est le ViewModel créé. Nous verrons une façon beaucoup plus propre d'écrire ce code plus loin.
Exécuter l'application. Cette fois le message s'affiche.

1.5. Codage du ViewModel

Chaque contrôle de la vue dont on souhaite récupérer la valeur ou la mettre à jour doit être lié à une property dans le ViewModel. Le lien se fait grâce à la clause `Binding` dans la vue XAML, comme vous en avez maintenant l'habitude.

Il s'agit du même code que celui du TD6 Partie 1, **à condition que vous ayez bien créé 3 properties**. **Seul changement : aucun code ne doit figurer dans le code behind de la vue XAML.** Ce code devra se trouver dans le ViewModel, ce qui pose problème pour les actions événementielles (clic sur un bouton, etc.). Voir section *Codage de l'action sur le bouton « Valider »* plus loin pour les indications à suivre.

Récupérer le code des properties associé aux 2 nombres à saisir et au résultat.

Codage de l'action sur le bouton « + » :

Nous allons gérer une commande sur le bouton « + ». En effet, avec le découpage View / ViewModel, le ViewModel n'est pas au courant d'une action sur l'interface, car c'est un fichier à part. Il n'est donc pas directement possible de réaliser une action dans le ViewModel lors d'un clic sur le bouton.
Les commandes correspondent à des actions faites sur la vue, comme un clic sur un bouton. Le XAML dispose d'un mécanisme simple de gestion de commandes via l'interface `IRelayCommand` (https://docs.microsoft.com/en-us/dotnet/api/microsoft.toolkit.mvvm.input.irelaycommand?view=win-comm-toolkit-dotnet-7.0). Par exemple, le contrôle `Button` possède (par héritage) une propriété `Command` du type `IRelayCommand` (https://docs.microsoft.com/en-us/windows/communitytoolkit/mvvm/relaycommand) permettant d'invoquer une commande lorsque le bouton est appuyé.
La classe `RelayCommand` permet ensuite de lier une commande à une action, i.e. une méthode.

a. Dans le fichier XAML, ajouter le code suivant :
```
<Button Content="+" …  Command="{Binding BtnSetAddition}"/>
```
b. Dans le fichier `CalculatriceVM`, ajouter la property suivante permettant de gérer le bouton (pas de méthode `set`) :
```
public IRelayCommand BtnSetAddition { get;}

public CalculatriceVM()
 {
     …

     //Boutons
     BtnSetAddition = new RelayCommand(ActionAddition);
 }

 public void ActionAddition()
 {
     //Code du calcul à écrire
 }
```

Using à ajouter : `CommunityToolkit.Mvvm.Input;`

Le bouton est maintenant créé. Il appelle une méthode nommée `ActionAddition` à coder. Cette méthode utilisera la classe `Calcul` **en utilisant** l'injection de dépendance comme dans le TD6 partie 1. Penser à ajouter une dépendance vers le projet `Calculatrice` et à installer le package Nuget `Microsoft.Extensions.DependencyInjection`.
Tester l'application.

1.6. Application de l'injection de dépendance

```
public partial class App : Application
{
    /// <summary>
    /// Gets the instance to resolve application services.
    /// </summary>
    public ServiceProvider Services { get; }

    public App()
    {
        /// <summary>
        /// Configures the services for the application.
        /// </summary>
        ServiceCollection services = new ServiceCollection();

        //Classes utiles
        services.AddSingleton<ICalcul, Calcul>();

        //ViewModels
        services.AddTransient<CalculatriceVM>();

        Services = services.BuildServiceProvider();
    }

    /// <summary>
    /// Gets the current app instance in use
    /// </summary>
    public new static App Current => (App)Application.Current;
}
```

Modifier le DataContext dans le code behind de la vue :

```
namespace WpfCalculatriceV3.Views
{
    /// <summary>
    /// Logique d'interaction pour CalculatriceView.xaml
    /// </summary>
    public partial class CalculatriceView : Window
    {
        public CalculatriceView()
        {
            InitializeComponent();
            DataContext = App.Current.Services.GetService<CalculatriceVM>();
        }
    }
```

Lancer l'application et la tester.

1.7. TaF

Coder le reste des fonctionnalités (soustraction,…, moyenne, factorielle).

Bilan

Le but premier du MVVM est de séparer les responsabilités, notamment en séparant les données de la vue. Cela facilite les opérations de maintenance en limitant l'impact d'éventuelles corrections sur un autre morceau de code.

Dans notre cas, nous avons pleinement appliqué le pattern MVVM car aucun code, hormis le DataContext, ne figure dans le fichier `CalculatriceView.xaml.cs`. Peu importe si vous ne respectez pas parfaitement le MVVM, le principe de ce pattern est de vous aider dans la réalisation de votre application et surtout dans sa maintenabilité.

L'intérêt également est qu'il devient possible de faire des tests unitaires sur le ViewModel, sans avoir besoin de réaliser des tests d'IHM. Cela permet de tester chaque fonctionnalité, dans un processus automatisé. Ce qui dans une grosse application est un atout considérable pour éviter les régressions de code…

***RAPPEL** - Etapes pour appliquer le patron d'architecture MVVM :*

1. *Installer le package NuGet CommunityToolkit.Mvvm*
2. *Créer une vue (fenêtre ou page) WPF*
3. *Créer une classe View Model héritant de* **`ObservableObject`**. Y créer les properties et les `IRelayCommand` nécessaires (1 par bouton). Les properties pourront appeler la méthode `OnPropertyChanged` en cas de mise à jour de la vue.
4. *Ajouter le binding sur les contrôles, y compris les boutons, dans la vue WPF.*
5. *Dans le constructeur du fichier App.xaml.cs enregistrer la classe du View Model créé en étape 3 dans le conteneur de services :*

```
public App()
{
    /// <summary>
    /// Configures the services for the application.
    /// </summary>
    ServiceCollection services = new ServiceCollection();

    //ViewModels
    services.AddTransient<CalculatriceVM>();
    services.AddTransient<MonNouveauViewModel>();

    Services = services.BuildServiceProvider();
}
```

6. *Dans le constructeur (code behind) de la vue WPF, ajouter le DataContext :*

```
DataContext = App.Current.Services.GetService<MonNouveauViewModel>();
```

2. ***Application du patron MVVM à l'application « Compte bancaire »***

Partir de la solution du TD6 – Partie 2.
Ajouter un nouveau projet `WpfComptesBancairesMVVM` de type WPF. Appliquer le patron MVVM pour gérer la fenêtre de retrait/dépôt.

3. ***Application du patron MVVM à l'application « Virement »***

Partir de la solution du TD6 – Partie 2.
Ajouter un nouveau projet `WpfVirementMVVM` de type WPF. Appliquer le patron MVVM pour gérer la fenêtre des virements.