**VSObserver principes de bases.**

L’application est programmé avec le Framework DotNet 4.0, en WPF (Windows Presentation Form) et en C#.

**Structure de l’application :**

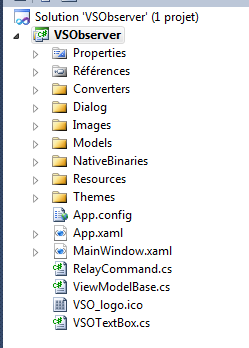


Figure 1 - Structure de l'application

La figure 1 présente l’architecture de l’application. Voici l’explication des éléments :

* Converters : Cet espace nom contient tous les convertisseurs, qui permettent de convertir des valeurs en élément graphique. Par exemple, il y a la classe « StringToColor » qui convertit une chaîne de caractère en couleur, pour l’afficher sur l’interface.
* Dialog : Cet espace nom contient tous les dialogues. Par exemple, il contient l’interface « ColoringRulesDialog.xaml », qui permet d’ajouter des règles de couleurs.
* Images : Dossier contenant toutes les images de l’application
* Models : Cet espace nom est le cœur de l’application. C’est lui qui contient toutes les classes permettant de réaliser les actions de VSObserver. J’y reviendrais plus en détails.
* NativeBinaries : Contient les librairies Libgit2Sharp, pour utiliser git à partir de C#.
* Resources : Certaines images de l’application y sont encore, à cause de l’ancienne architecture. Il y a également le fichier « Rule.xsd » qui permet de vérifier la conformité du fichier XML de colorisation.
* Themes : Permet de personnalisé certains éléments graphique.
* App.config : Fichier de configuration. Il permet de spécifier le taux de rafraîchissement de l’application par exemple. Ce fichier est accessible à tous et l’application met à jour les modifications au démarrage.
* App.xaml : Spécification des ressources graphiques
* MainWindow.xaml : Fenêtre principale de VSObserver
* RelayCommand.cs : Classe qui permet de réaliser des commandes. Par exemple, exécuté une action dans l’une des classe du « Models », avec un bouton.
* ViewModelBase.cs : Classe générique permettant d’implémenter un parton MVVM. Cette classe est utilisée pour mettre en relation l’interface graphique et les classes présentent dans l’espace nom « Models ».
* VSO\_logo.ico : Logo de l’application
* VSOTextBox.cs : Personnalisation de la TextBox pour la recherche

**Présentation d’une interface graphique :**

Les interfaces graphiques sont réalisées en XML. Pour présenter le concept du MVVM, je vais partir d’un exemple (recherche). L’interface graphique de la recherche est en fait un TextBox personnalisé. La classe « Generic.xaml » dans le dossier « Themes » permet de spécifier le comportement graphique. Comme la couleur de fond, une couleur spécifique si il y a une erreur, etc. Pour personnalisé le TextBox de recherche, il faut également définir les attributs de l’élément graphique. Pour ce faire la classe « VSOTextBox .cs » est implémentée. Cette classe décrit le comportement de l’objet. Par exemple, il y a l’attribut « TotalVariable » qui permet d’afficher le nombre total de variables (voir figure 2).

Nombre total des variables



Figure 2 - Champ recherche qui correspond à un TextBox personnalisé; Nombre total des variables

L’attribut « NbElement » va permettre de récupérer la valeur envoyé à partir du XML. La propriété « NbElementProperty » va permettre de pouvoir mettre dynamiquement le nombre de variables trouvées. Le code ci-dessous présent dans « VSOTextBox.cs » permet de définir un champ personnalisé dans une TextBox. On va pouvoir récupérer ce champ lorsqu’on définira notre TextBox personnalisé.

public static readonly DependencyProperty NbElementProperty =

DependencyProperty.Register("NbElement", typeof(int), typeof(VSOTextBox), new PropertyMetadata(0));

/// <summary>

/// Nombre d'éléments trouvés

/// </summary>

public int NbElement

{

get { return (int)GetValue(NbElementProperty); }

set { SetValue(NbElementProperty, value); }

}

Pour pouvoir utiliser le TextBox dans la vue principale, il faut déclarer l’espace de nom en XML comme le code suivant le montre :

Pour déclarer un espace nom en XML, il faut toujours commencer par xmlns: suivi du nom que l’on veut. Dans notre cas le nom est « nspace ». Ensuite, on déclare entre quotte clr-namespace : suivi de l’espace nom présent dans la classe « VSOTextBox.cs ».

Nom de l’objet graphique. Dans notre cas c’est une fenêtre. Il faut toujours que la balise soit fermé. Soit par <Window /> ou <Window></Window>.

<Window x:Class="VSObserver.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:nspace="clr-namespace:VSObserver"

xmlns:converters="clr-namespace:VSObserver.Converters"

Title="VSObserver" Height="400" Width="550"

SizeToContent="Manual"

Background="{StaticResource WindowBackgroundColor}" Icon="/VSObserver;component/Images/VSO\_logo.ico">

Une fois l’espace de noms déclaré il faut ajouter notre TextBox personnalisé dans la fenêtre. Pour ce faire on va ajouter notre balise XML à l’intérieur d’un « Grid » qui est à l’intérieur de « Window ». Le « Grid » est un élément permettant de disposer plusieurs éléments graphiques sous forme de tableau. Voici des informations supplémentaires concernant les agencements (layout) en WPF : <http://wpftutorial.net/Layout.html> .

Le code ci-dessous présente l’ajout de la TextBox personnalisé dans la fenêtre :

Notre TextBox personnalisé précédé de l’espace de noms. Le nom correspond au nom de la classe.

<nspace:VSOTextBox x:Name="tb\_variableName" AcceptsReturn="False" Margin="5,5,5,5" Height="24" Width="Auto" HorizontalAlignment="Stretch"

Grid.Column="0" Grid.Row="0"

On retrouve notre champ personnel décrit plus haut.

Text="{Binding Path=SearchText, UpdateSourceTrigger=PropertyChanged}"

TextChanged="tb\_variableName\_TextChanged"

NbElement="{Binding VarNumberFound, UpdateSourceTrigger=PropertyChanged}"

SearchError="{Binding InformationMessage, UpdateSrrrourceTrigger=PropertyChanged}"

IsEnabled="{Binding SearchTextEnabled}"/>

Attribution du texte à la TextBox. On réalise une liaison (binding). Cet élément nous permet à chaque changement de mettre à jour l’attribut « SearchText ». Le « binding » arrive à déterminer que c’est le bon champ, car son DataContext contient ce champ. Le champ « UpdateSourceTrigger=PropertyChanged » permet de spécifier quand on met à jour la valeur (une fois, lorsqu’on perd le focus, à chaque changement…).

Pour attribuer un DataContext il faut aller dans le code de CSharp de la fenêtre graphique (clique droit > afficher le code sur le fichier MainWindow.xaml). Une fois dans le code, il faut déclarer la ligne suivante pour attribuer un DataContext :

///Création d’un objet VariableObserver. Cette classe représente le cœur de l’application c’est elle qui va faire la recherche, la mise à jour des couleurs, les actions (forçages / écritures)…

vo = new VariableObserver(ipAddresseRTCServer, sqlLiteDataBase, number\_variable);

///Attribution du datacontext de la textbox (voir le champ x:Name)

tb\_variableName.DataContext = vo;

La classe « VariableObserver » hérite de la classe « ViewModelBase », ainsi ça permet de pouvoir liés de champ dans le XAML. En conséquence, lorsque un utilisateur tape du texte il sera mis à jour dans la classe « VariableObserver ». Le code ci-dessous montre un exemple.

class VariableObserver : ViewModelBase

{

///Constante qui permet de spécifier quel champ mettre à jour

private const string SEARCH\_TEXT = "SearchText";

///Notre variable qui contiendra la recherche

private string \_searchText;

///Champ SearchText définit dans le XAML

public string SearchText

{

get { return \_searchText; }

set

{

///Lorsqu’il y a un changement la valeur est mise dans la variable \_searchText, la méthode OnPropertyChanged va mettre le champ spécifié à jour. En clair elle va appeler le get définit plus haut pour mettre à jour la valeur.

\_searchText = value;

OnPropertyChanged(SEARCH\_TEXT);

}

}

}

Enfin, la classe « DataObserver » contient les éléments de base de VS puis certains éléments rajouté, comme la couleur. Cette classe correspond à une ligne dans le tableau dans VSObserver. A chaque recherche effectuée dans VariableObserver, la méthode rechercher va créer une liste de « DataObserver ». Cette liste sera affichée dans le tableau dans VSOBserver. La figure 3 présente un exemple d’objet « DataObserver ».



Figure 3 - Représentation d’un « DataObserver »