# Comment tirer efficacement parti des avantages de WPF tout en gardant une application maintenable et testable ?

Par Jérémy Alles

Date de publication : 12 mars 2009

Petit à petit, la technologie WPF est adoptée par les développeurs .Net comme plateforme de développement pour les applications graphiques de prochaine génération. Cette transition est annoncée comme longue et difficile, car WPF remet en cause un certain nombre de principes jusque là bien établis pour la création d'interfaces graphiques utilisateur. La méthodologie MVVM permet de formaliser le développement avec WPF en garantissant des applications bien architecturées, testables et optimisant le flux de travail entre développeur et designer.

**Commentez**

# I - Préambule

Dans cet article, je présente une méthodologie, appelée MVVM pour Modèle-Vue-VueModèle. MVVM permet de tirer partie des bénéfices de la plateforme WPF tout en conservant une application correctement architecturée, maintenable et testable. Vous remarquerez que tout au long de l'article, j'utilise plus le terme méthodologie que design pattern pour parler de MVVM. Ce choix est personnel, et les ressources que l'on trouve sur le sujet, notamment en anglais, utilisent parfois la deuxième terminologie.

J'estime que le lecteur connaît les fondamentaux de WPF (XAML, Databinding, Commandes, etc.) et de la plateforme .Net. C'est le langage C# qui sera utilisé dans les exemples de code. A l'issue de la lecture de ce document, le lecteur sera familiarisé avec les principes de la méthodologie MVVM et sera en mesure de l'appliquer pour ses propres projets.

**Remarque :** les principes décrits dans cet article sont destinés à être utilisés avec l'environnement WPF. La version « Web », Silverlight, étant en certains points différente de WPF, des aspects ne peuvent pas être appliqués sans modifications. Néanmoins, les idées générales sont valables pour les deux plateformes.

# II - Introduction

Aujourd'hui, produire un logiciel de qualité nécessite différents aspects :

- Adéquations avec les attentes des utilisateurs
- Apparence visuelle
- Manière d'interagir avec l'utilisateur
- Performances

Pour atteindre ces objectifs, des compétences variées doivent collaborer:

- Un maître d'oeuvre pour définir les besoins des utilisateurs
- Un designer pour créer des images et des animations
- Un ergonome pour s'assurer que le contenu est présenté à l'utilisateur de la bonne manière
- Un architecte pour organiser l'application
- Un développeur pour intégrer le travail du designer et réaliser les fonctionnalités attendues
- Etc.

Retrouver toutes ces compétences au sein d'une même personne est difficile et improbable. Le développement logiciel nécessite donc l'intervention de personnes différentes, que l'on peut qualifier à plus haut niveau de développeur (architecte, testeur, codeur, etc.) et designer (artiste, ergonome, animateur, etc.).

Le souhait de vouloir combiner le travail des développeurs et des designers n'est pas quelque chose de nouveau, mais semblait jusque là un objectif difficile à atteindre. Aujourd'hui, avec la plateforme WPF, un accent particulier à été mis sur cette collaboration afin de simplifier la création de logiciels de nouvelle génération graphique.

La clé de cet échange entre designer et développeur repose sur XAML. C'est au travers de ce langage à balises que les deux mondes partagent leurs travaux pour la réalisation du logiciel. Cette collaboration est également renforcée par l'utilisation d'une suite logicielle adaptée :

- Visual Studio 2008 pour les développeurs
- La suite Expression (Blend, Design) pour les designers.

Les principes décrits dans MVVM ont pour but de maximiser cette collaboration en produisant un logiciel de qualité, maintenable et testable. Le dernier point est particulièrement important, car, comme nous allons le voir dans la suite, il est facile d'écrire beaucoup de code qui fait ce que l'on veut, mais qui sera difficile à tester.

En résumé, la méthodologie MVVM peut apporter les bénéfices suivants :

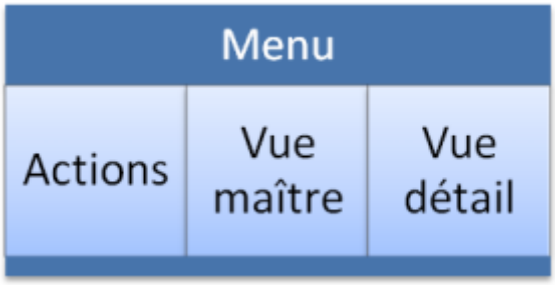
- Une séparation claire entre le code métier et sa représentation graphique
- La possibilité d'utiliser efficacement et simplement les mécanismes de WPF (databinding, commandes, etc.)
- La production d'un code qui est testable
- Une organisation qui facilite et optimise le flux de travail entre développeur et designer
- Un déploiement facilité (réutilisation du travail) en multi-environnement (WPF/Silverlight)

## III - Architecture classique d'une application WPF

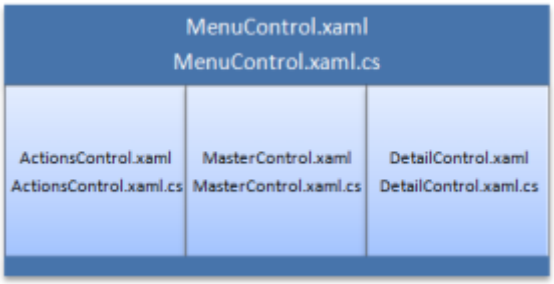
Lorsque l'on commence à travailler avec WPF, on réalise souvent des prototypes. WPF offre en effet l'avantage de pouvoir tester très rapidement une fonctionnalité en réalisant une maquette.

Quand arrive l'heure de la réalisation, on commence généralement par une simple fenêtre vide. C'est dans cette fenêtre que l'on va venir ajouter petit à petit les contrôles nécessaires à la réalisation de notre interface. Quand il le faut, le fichier « code-behind » de la fenêtre (.xaml.cs) est complété afin de rajouter du code lors de l'initialisation, de répondre à des événements, etc.

Le développement de notre application progressant, on éprouve assez rapidement le besoin d'organiser un peu mieux notre projet. Une possibilité est de découper notre fenêtre en zones, chacune ayant un objectif précis pour l'utilisateur. Par exemple :




Afin d'architecturer notre fenêtre, on réalise un UserControl pour chaque zone. Cela permet de découper logiquement notre fenêtre en éléments et de faciliter le développement et la maintenance. Les UserControls sont alors « assemblés » dans la fenêtre principale (dans l'exemple précédent, on pourra placer nos quatre UserControls dans un DockPanel).




Afin d'utiliser le DataBinding entre le XAML et le code C#, il est nécessaire de définir la propriété DataContext. Une possibilité est de définir le DataContext sur l'instance du UserControl, puis d'exposer dans le code behind les propriétés nécessaires dans le XAML :


```
<CheckBox IsEnabled="{Binding IsEnabled}"/>
```

```
public partial class ActionsControl : UserControl
{
    public bool IsEnabled
    {
        get; set;
    }

    public ActionsControl()
    {
        InitializeComponent();

        this.DataContext = this;
    }
}
```

La solution décrite ici est entièrement fonctionnelle, néanmoins, certaines limitations peuvent être notées :

- Le flux de travail développeur / designer est brisé ou difficile à maintenir

Si les rôles sont attribués à deux personnes distinctes (comme cela serait le cas idéalement), le designer modifie le XAML et le développeur le code-behind. Les deux fichiers étant très fortement couplés, les évolutions sont difficiles. C'est particulièrement vrai pour les gestionnaires d'événements qui peuvent être fastidieux à manipuler pour le designer.

- L'écriture des tests est fastidieuse

Il faut en effet être capable d'instancier le UserControl en question. Les tests sont ensuite difficiles à écrire car il faut être capable de simuler des entrées en provenance de l'interface graphique (clic sur un bouton ou saisie dans un formulaire par exemple).

- Les fichiers code-behind sont difficilement maintenables

Ces fichiers peuvent devenir très volumineux en nombre de lignes de code. De plus, il n'y a aucune abstraction entre la représentation graphique des données (dans le XAML) et les données réelles (dans le code-behind et dans les classes métier).

- Il est difficile de réutiliser du code déjà existant qui fonctionnait avant WPF

Si l'on doit réutiliser une librairie, il peut être difficile de visionner la séparation entre l'interface graphique et le code déjà existant. En utilisant directement la librairie, on pourra être tenté de la modifier pour la rendre compatible avec les notions utilisées par WPF.

- Il est difficile de manipuler certains contrôles WPF

WPF a été conçu avec cet objectif de séparation entre le comportement d'un contrôle (par exemple le fait de sélectionner un item au sein d'une liste) et son rendu graphique (pour un ListBox, une zone d'affichage avec un ascenseur sur la droite du contrôle). Cette séparation permet une souplesse inouïe mais parfois avec un niveau de difficulté plus accrue. L'exemple le plus clair est probablement la gestion d'un TreeView.

En utilisant le contrôle TreeView et une source de données, la génération des TreeViewItem est entièrement automatique (comme c'est le cas pour tous les contrôles dérivant d'ItemsControl (1) ). Idéalement, il ne faudrait jamais avoir à manipuler (par exemple pour gérer la propriété IsSelected) directement les TreeViewItem depuis le

code (ou les ListBoxItem) car ces contrôles sont purement graphiques. Sans une abstraction entre la représentation graphique et les données, c'est malheureusement bien souvent le cas (en particulier en jouant avec la classe ItemContainerGenerator).

La méthodologie MVVM a pour but de résoudre les problèmes énoncés ici. Avant de détailler les points pratiques de MVVM, faisons d'abord un court détour par les notions de Modèle, de Vue et de Contrôleur...

## IV - Modèle, Vue, Contrôleur

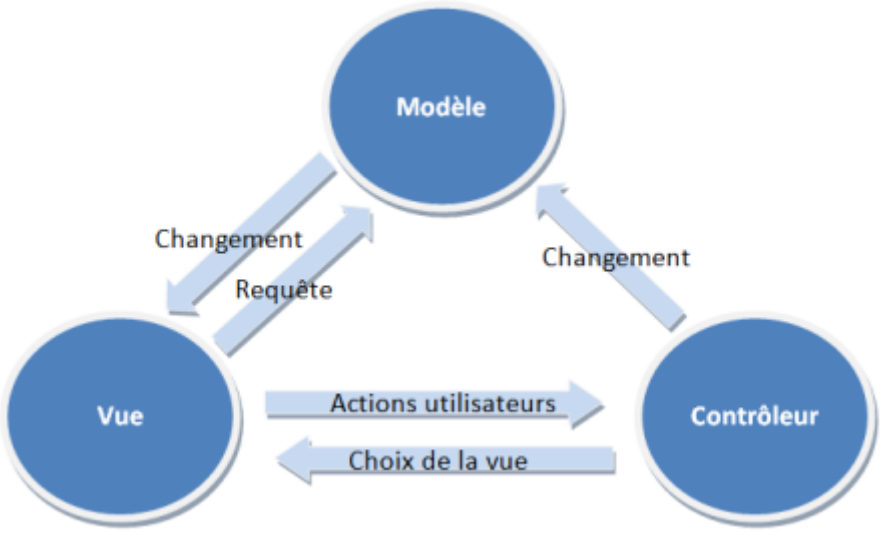
Comme nous venons de le voir dans la partie précédente, la mise au point de l'architecture d'une interface graphique peut se révéler délicate et complexe. Une première approche est d'extraire de l'application un Modèle. Le modèle contient les données primitives que l'application manipule, spécifiques au domaine de l'application.

L'intérêt d'extraire le Modèle dans l'application est multiple :

- Organisation plus claire de l'application
- Réutilisation d'un modèle déjà existant, validé et testé
- Développement du modèle par une équipe dédiée
- Indépendance vis-à-vis de la représentation graphique associée

Si vous avez déjà travaillé à la conception d'un logiciel utilisant une interface graphique utilisateur, alors il ne fait presque aucun doute que vous avez déjà croisé le terme MVC. Derrière cet acronyme se cache un patron de conception divisant l'IHM en un Modèle (modèle de données), une Vue (présentation des données à l'utilisateur) et un Contrôleur (gestion des événements).

Le diagramme suivant présente l'architecture MVC, ses 3 blocs et les relations entre chaque entité.




Lorsque l'utilisateur sollicite l'interface graphique, les actions suivantes sont exécutées :

1. La requête est analysée par le contrôleur
2. Le contrôleur demande au modèle d'effectuer les traitements
3. Le contrôleur renvoie le vue adaptée si le modèle ne l'a pas déjà fait

Il est possible d'utiliser MVC avec WPF. Néanmoins, du fait des nouveautés de la plateforme en termes de fonctionnalités (databinding, commande) comme en termes de flux de travail (développeur et designer travaillant conjointement), l'architecture peut être revisitée pour s'adapter à la puissance de WPF. C'est justement l'objet de la méthodologie MVVM décrite dans la partie suivante.

# V - Présentation de la méthodologie MVVM

## V-A - Définition

La méthodologie Modèle / Vue / Vue-Modèle est une variation du patron de conception MVC, taillée sur mesure pour les technologies modernes d'interface utilisateur où la réalisation de la Vue est davantage confiée à un designer qu'à un développeur classique. Le terme designer désigne ici un artiste davantage concerné par les aspects graphiques et d'expérience utilisateur que par l'écriture de code. Le design en question est réalisé en langage déclaratif (ici XAML) à l'aide d'outils spécialisés (Expression Blend).

## V-B - Modèle

Le modèle est tel qu'il est défini dans MVC : il constitue la couche de données métier et n'est lié à aucune représentation graphique précise. Le modèle est réalisé en code C# et ne repose pas sur l'utilisation de technologie propre à WPF. Le Modèle n'a aucune connaissance de l'interface graphique qui lui est associée.

## V-C - Vue

La vue est constituée des éléments graphiques : fenêtre, bouton, sélecteur, etc. Elle prend en charge les raccourcis clavier et ce sont les contrôles eux-mêmes qui gèrent l'interaction avec les périphériques de capture (clavier / souris) alors que c'était la responsabilité du Contrôleur dans MVC. Les contrôles sont rassemblés dans un UserControl, dont le nom est généralement suffixé par « View » (par exemple MasterView.xaml).

Pour l'utilisation et la présentation des données à l'utilisateur, le DataBinding entre en jeu. Dans des cas simples, la Vue peut être directement « bindée » sur le Modèle : par exemple le modèle expose une propriété Booléenne qui est bindée sur un CheckBox dans la Vue.

En revanche, pour des cas plus complexes, le Modèle peut ne pas être directement exploitable par la Vue. Cela peut être le cas lors de la réutilisation d'un Modèle déjà existant qui ne présente pas les propriétés attendues par la Vue.

## V-D - Vue-Modèle

C'est justement la partie Vue-Modèle qui est responsable des tâches énoncées précédemment. Le bloc Vue-Modèle peut être vu comme un adaptateur entre la Vue et Modèle, qui réalise plusieurs tâches :

- Adaptation des types issus du Modèle en des types exploitables via le databinding par la Vue
- Exposition de commandes que la vue utilisera pour interagir avec le Modèle
- Garde trace de la sélection courante des contrôles

Les classes Vue-Modèle exposent donc des propriétés qui seront directement exploitables depuis le XAML définissant la Vue. Ces propriétés peuvent n'avoir qu'un accesseur get dans le cas d'un binding OneWay, ou des accesseurs get et set pour un binding TwoWay. Dans les deux cas, l'implémentation de l'interface **INotifyPropertyChanged** est nécessaire afin de signaler les changements au moteur de binding de WPF.

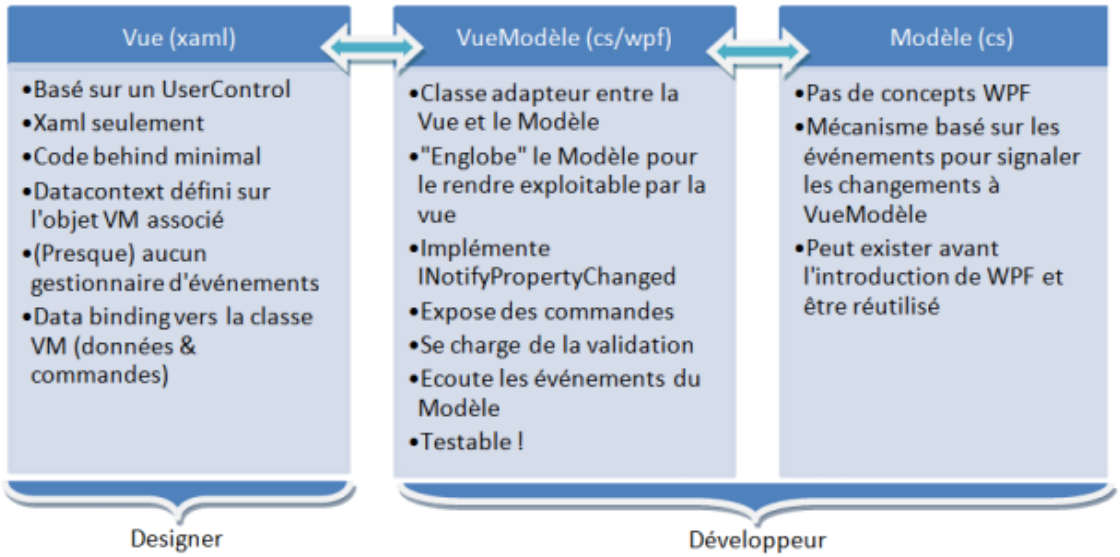
En règle générale, les classes Vue-Modèle n'ont pas besoin de connaître la Vue qui les exploite. Parfois, il peut être nécessaire de connaître cette information, mais des mécanismes existent pour essayer d'abstraire au maximum les classes Vue-Modèle des Vues associées (avec l'utilisation d'interfaces par exemple).

Les classes Vue-Modèle introduisent donc une abstraction entre la partie graphique (le XAML, réalisé par le designer) et le reste (les classes Vue-Modèle et Modèle réalisées par les développeurs). Le but est d'obtenir une architecture la plus faiblement couplée possible.

En plus des fonctionnalités de WPF qui rendent l'utilisation de la méthodologie complètement naturelle (Databinding, DataTemplate, etc.), les classes Vue-Modèle offrent l'avantage d'être faciles à tester. Puisque les classes Vue-Modèle sont des classes "standard", il devient possible d'écrire des tests unitaires. En quelque sorte, les tests unitaires et la vue (en XAML) deviennent tous deux consommateurs de la classe Vue-Modèle associée.

## V-E - Collaboration entre designer et développeur

Le diagramme suivant résume le fonctionnement des différents blocs:




## VI - Principes de la méthodologie MVVM

Dans ce paragraphe, je présente les grands principes qui sont applicables à la méthodologie MVVM. Chacun de ces principes sera illustré par une application exemple présentée dans la suite de l'article.

## VI-A - Classe de base pour les classes Vue-Modèle

Puisque toutes les classes Vue-Modèle doivent implémenter l'interface INotifyPropertyChanged, ce travail peut n'être fait qu'une fois, en définissant une classe de base dont toutes les classes Vue- Modèle dériveront. J'ai l'habitude pour cela d'écrire une méthode protégée OnPropertyChanged qui attend en argument le nom de la propriété qui a changée, et qui lève l'événement PropertyChanged associé.

De plus, une idée que j'ai reprise sur un article de Josh Smith (2) consiste à vérifier (en mode debug seulement), que la propriété pour laquelle l'événement PropertyChanged est levé existe bien (en utilisant la réflexion). Cela signifie qu'un appel tel que OnPropertyChanged(« Nom ») produira une exception visible au débuggeur sur la propriété Nom n'existe pas. Cette petite astuce permet d'éviter certaines erreurs de binding.

## VI-B - Instanciation des classes Vue-Modèle

Lorsqu'une Vue est créée, il est nécessaire de créer la classe Vue-Modèle qui lui est associée. Il est possible de réaliser cela de deux manières :

- Depuis le code-behind en C#, on instancie la classe Vue-Modèle dans le constructeur et on définit le DataContext de la Vue

```
public ActionsView()
```

```
{
    InitializeComponent();
    this.DataContext = new ActionsViewModel();
}
```

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

```
<UserControl.DataContext>
    <local:ActionsViewModel/>
</UserControl.DataContext>
```

Les deux méthodes sont complètement équivalentes. A titre personnel, je préfère utiliser la première pour pouvoir éventuellement fournir un argument au constructeur de la classe Vue-Modèle.

## VI-C - Databinding

Une fois que la classe Vue-Modèle est instanciée et que le DataContext de la Vue est positionné convenablement, il est possible de réaliser des bindings entre la Vue et les propriétés du Vue- Modèle.

Le fait de disposer d'une classe adaptateur entre la Vue et le Modèle peut permettre dans certains cas de se passer de convertisseur (3) (IValueConverter ou IMultiValueConverter). En effet, la classe Vue- Modèle ayant justement le rôle de présenter les données en vue de leur affichage, le travail de conversion peut être effectué ici.

L'utilisation de l'interface INotifyPropertyChanged permet de synchroniser un binding avec une propriété. Dans le cas d'un binding vers une liste d'objets, il faut implémenter l'interface INotifyCollectionChanged. Fort heureusement, WPF fournit sa propre implémentation : c'est la fameuse classe ObservableCollection.

ObservableCollection est une nouvelle classe du framework 3.x. De ce fait, il est tout à fait possible de ne pas souhaiter utiliser cette classe dans le Modèle, ou bien de réutiliser un Modèle antérieur au framework 3.x (qui n'utilise donc pas d'ObservableCollection).

Pour cela, lors de l'initialisation de la classe Vue-Modèle, la liste d'objets gérée par le Modèle est parcourue pour remplir une ObservableCollection. Les événements « élément ajouté » et « élément supprimé » du Modèle sont gérés par la classe Vue-Modèle afin de mettre à jour l'ObservableCollection en conséquence.

## VI-D - Utilisation des commandes

WPF introduit la notion de commande au travers de l'interface ICommand. Les implémentations fournies avec WPF de cette interface sont les classes RoutedCommand et RoutedUICommand. Nous allons voir dans cette partie que ces implémentations ne sont pas adaptées à une utilisation avec la méthodologie MVVM. Nous verrons pourquoi et comment utiliser le mécanisme des commandes dans notre cas.

La documentation MSDN associée à cette notion donne l'introduction suivante :
*« Contrairement à un simple gestionnaire d'événements joint à un bouton ou une minuterie, les commandes séparent la sémantique et l'appelant d'une action de sa logique. Cela permet à des diverses sources d'appeler la même logique de commande et de personnaliser la logique de commande par rapport à différentes cibles. »*

Si vous avez déjà utilisé les RoutedCommand, vous avez dû remarquer que leur utilisation dans une application est quelque peu fastidieuse. La commande est définie statiquement dans une classe, et afin d'associer une action à une commande, il faut passer par la collection CommandBindings du contrôle auquel est associée la commande :

```
this.CommandBindings.Add(
    new CommandBinding(
        ApplicationCommands.Find,
```

```
        Find_Execute,
        Find_CanExecute));
```

C'est en particulier l'utilisation des CommandBindings (qui imposent d'ajouter du code dans le fichier .xaml.cs) qui n'est pas compatible avec la méthodologie MVVM. Dans notre cas, on aimerait avoir les points suivants :

- Définition de la commande dans la classe Vue-Modèle: une commande est un moyen que le développeur offre au designer pour interagir avec le Modèle
- Gestion des actions Execute/CanExecute dans la classe Vue-Modèle : c'est également la responsabilité du développeur de décider quoi faire quand la commande est exécutée, et quand elle peut l'être

Cela est possible, en réalisant notre propre implémentation de l'interface ICommand. Vous verrez dans l'application d'exemple comment cela est réalisé, et vous découvrirez à quel point il devient simple et naturel d'utiliser les commandes !

## VI-E - Gestion de la sélection utilisateur

La question ici est de connaître, dans les classes Vue-Modèle, l'état de la sélection courante de l'utilisateur. La « sélection » s'applique ici aux contrôles qui dérivent de la classe Selector, c'est-à-dire aux ComboBox, ListBox et TabControl.

Une approche est d'utiliser la propriété SelectedIndex qui peut répondre à ce besoin. Une autre possibilité beaucoup plus puissante est d'utiliser l'interface ICollectionView. Au travers d'une méthode statique de la classe CollectionViewSource, il est très facile, à partir d'une source de données bindée dans la Vue, d'obtenir l'interface ICollectionView associée.

```
ICollectionView collectionView = CollectionViewSource.GetDefaultView(this.Collections);
```

Cette interface permet également d'observer la sélection courante du contrôle associé à la source de données. Pour cela, il est important de définir la propriété IsSynchronizedWithCurrentItem à True. ICollectionView expose en effet une propriété CurrentItem qui, comme son nom l'indique, contient une référence vers l'objet actuellement sélectionné.

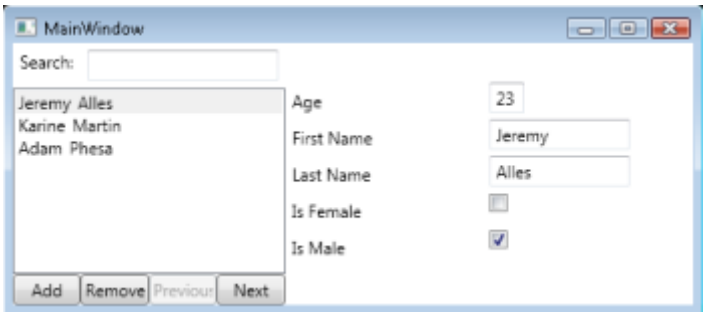
Cette fonctionnalité permet notamment de construire des vues maître/détail facilement. Enfin, il est également possible d'utiliser ICollectionView pour réaliser des opérations de tri, de filtrage ou de regroupement (en utilisant simplement les propriétés Sort, Filter et Group).

## VII - Exemple

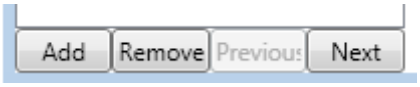
Afin d'illustrer les principes décrits dans la section précédente, nous allons explorer une application très simple utilisant la méthodologie MVVM.
Vous pouvez télécharger les sources de l'application **ici** (**miroir**)

## VII-A - Présentation générale

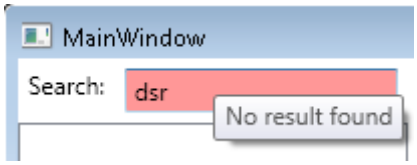
Cet exemple a été réalisé rapidement, en utilisant Visual Studio 2008 et sans aucun travail sur la partie graphique (je suis personnellement plus développeur que designer !). Etant à court d'idée originale, le principe de l'application est également minimaliste : maintenir une liste de personne (nom, prénom, sexe, âge) et pouvoir éditer cette liste. La fenêtre se présente comme suit :




La partie gauche donne une vue des personnes disponibles dans la liste (3 personnes sont ajoutées automatiquement au démarrage). La sélection d'une personne permet de l'éditer dans la vue détail associée (sur la droite). Des boutons permettent : l'ajout ou la suppression d'une personne, la navigation dans la liste (personne suivante et précédente). Les boutons sont reliés à des commandes ce qui permet de les désactiver automatiquement en fonction du contexte courant de l'application (on ne peut pas exemple pas avoir plus de 5 personnes dans la liste) :




Il est également possible de chercher une personne par son nom ou son prénom (la recherche et sensible à la casse et cherche sur les premières lettres). Le champ de recherche signale un résultat vide par un style particulier :




L'application utilise les principes suivants :

- Décomposition de la fenêtre en 2 UserControl (MasterView et DetailView)
- Instanciation des classes Vue-Modèle depuis le constructeur
- Utilisation de l'interface ICollectionView pour la recherche et la navigation
- Utilisation d'une classe ViewModelBase comme classe mère de toutes les classes Vue-Modèle
- Utilisation d'une implémentation de l'interface ICommand permettant la définition des commandes entièrement dans les classes Vue-Modèle

## VII-B - Gestion des commandes

Les commandes sont exposées sous forme de propriété dans les classes Vue-Modèle et utilise une implémentation de ICommand nommée RelayCommand. L'utilisation est très simple :

```
public ICommand AddCommand
{
    get
    {
        if (this.addCommand == null)
            this.addCommand = new RelayCommand(() => this.AddPerson(), () =>
this.CanAddPerson());

        return this.addCommand;
    }
}
```

Dans l'accesseur get, si la commande n'est pas encore créée, alors on l'instancie. Pour cela, on crée une instance de la classe RelayCommand dont le prototype est le suivant :

```
public RelayCommand(Action execute, Func<bool> canExecute)
```

L'utilisation d'expressions Lambda (avec la syntaxe =>) permet de créer facilement les délégués Action et Func<bool> Les fonctions associées sont également assez simples : la méthode CanAddPerson vérifie que le nombre de personnes est inférieur à 5, la méthode Add utilise la méthode AddPerson pour crée le nouvel objet Person et l'englober dans un PersonViewModel.

```
private bool CanAddPerson()
{
    return this.persons.Count < 5;
}

private void AddPerson()
{
    Person newPerson = this.db.AddPerson("firstName", "lastName", 25, Gender.Female);
    this.persons.Add(new PersonViewModel(newPerson));
}
```

## VII-C - Recherche

Le contrôle de recherche (un TextBox) a sa propriété Text bindée sur la propriété SearchText de la classe Vue-Modèle associé. A chaque set de la propriété, on met à jour la propriété Filter de l'objet ICollectionView :

```
public string SearchText
{
    set
    {
        this.collectionView.Filter = (item) =>
         {
            if (item as PersonViewModel == null)
                return false;

            PersonViewModel personViewModel = (PersonViewModel) item;
            if (personViewModel.FirstName.Contains(value) ||
                personViewModel.LastName.Contains(value))
                return true;

            return false;
         };

        this.OnPropertyChanged("SearchContainsNoMatch");
    }
}
```

Le set signale également le changement de la propriété SearchContainsNoMatch qui permet de savoir si la recherche contient des résultats. Cette propriété est utilisée par un style du TextBox afin de changer l'apparence du contrôle quand la recherche est vide :

```
<TextBox Text="{Binding SearchText, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" Margin="5">
  <TextBox.Style>
    <Style>
      <Style.Triggers>
        <DataTrigger Binding="{Binding SearchContainsNoMatch}" Value="True">
          <DataTrigger.Setters>
            <Setter Property="TextBox.Background" Value="#68FF0000"/>
            <Setter Property="TextBox.ToolTip" Value="No result found"/>
          </DataTrigger.Setters>
        </DataTrigger>
      </Style.Triggers>
    </Style>
  </TextBox.Style>
```

```
</TextBox>
```

## VII-D - Suppression d'une personne

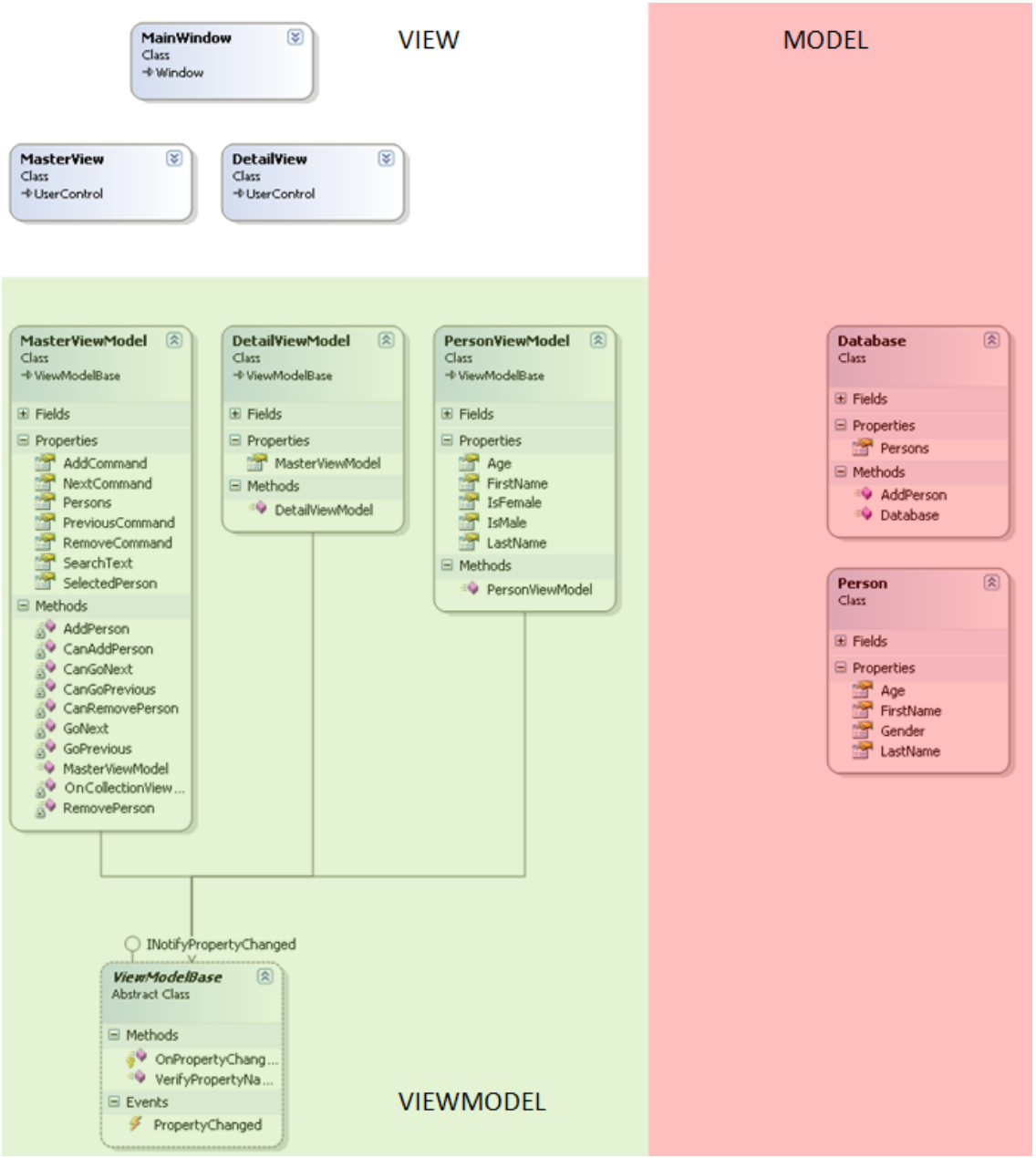
L'implémentation de la suppression d'une personne est minimaliste :

```
private void RemovePerson()
{
    this.db.RemovePerson(this.SelectedPerson.Person);
        this.persons.Remove(this.SelectedPerson);
}
```

La personne est tout d'abord supprimée de la « database », puis on supprime l'objet Vue-Modèle associé de l'ObservableCollection. Cette méthode pourrait être améliorée avec un retour de la méthode DataBase.RemovePerson pour savoir si la suppression s'est bien effectuée si ce n'est pas le cas, il ne faudrait pas supprimer l'objet Vue-Modèle.

## VII-E - Diagramme de classes

Le diagramme suivant montre la décomposition de l'application entre les classes Modèles, Vue et Vue-Modèle.




## VIII - Aller plus loin

Quelques liens à explorer pour en savoir plus sur la méthodologie MVVM (en anglais) :

- Karl a débuté une série sur MVVM, avec des exemples très bien faits et expliqués (code source en VB): **http://karlshifflett.wordpress.com/mvvm/**
- Karl et Josh ont écrit un article sur CodeProject qui démontre les principes de MVVM : **http://karlshifflett.wordpress.com/mvvm/internationalized-wizard-in-wpf-using-m-v-vm/**
- Karl a écrit un article à propos de la validation de données avec MVVM : **http://karlshifflett.wordpress.com/mvvm/input-validation-ui-exceptions-model-validation-errors/**
- Josh a écrit un article à propos de MVVM pour la gestion d'un TreeView : **http://www.codeproject.com/Articles/26288/Simplifying-the-WPF-TreeView-by-Using-the-ViewMode**
- Marlon a écrit un article à propos de l'interface ICollectionView : **http://marlongrech.wordpress.com/2008/11/22/icollectionview-explained/**

## IX - Conclusion

Au fil de cet article, j'ai essayé de montrer comment la méthodologie MVVM peut permettre de faciliter la réalisation d'une application WPF. Evidemment, pour les interfaces graphiques très simples, le surcoût de la méthodologie MVVM peut ne pas valoir le coup. Dans les autres cas, les avantages sont nombreux :

- Abstraction de la vue
- Réduction du code dans le code-behind
- Les classes Vue-Modèle sont testables simplement
- Amélioration du flux de travail développeur / designer

Les ressources sur le web sont nombreuses, et la méthodologie, tout comme la plateforme WPF est encore toute jeune. Des bloggeurs discutent régulièrement leurs aventures avec MVVM ou comment la méthodologie aide à la résolution d'un problème concret.

Je remercie chaleureusement toutes les personnes qui ont aidées à la publication de cette article. Mes relecteurs: Aline, Charlotte, Fred et TomLev. Louis-Guillaume de l'équipe .Net de developpez.com qui m'a assisté tout au long de la publication.

Je suis ouvert à tout commentaire, remarque ou suggestion pour améliorer cet article. Pour cela, n'hésitez pas à me contacter !

## X - Références

- The New Iteration : How XAML Transforms the Collaboration Between Developers and Designers in WPF, Kartsen Janusewsky and Jaimes Rodriguez (**http://msdn.microsoft.com/en-us/library/vstudio/ms754130.aspx**)
- Introduction to Model/View/ViewModel pattern for building WPF apps, John Gossman (**http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx**)
- WPF apps with the Mode-View-ViewModel patterns, Josh Smith (**http://msdn.microsoft.com/en-us/magazine/dd419663.aspx**)
- ICollectionView explained, Marlon Grech (**http://marlongrech.wordpress.com/2008/11/22/icollectionview-explained/**)

1 : On pourra consulter l'excellente série " ItemsControl : A to Z " de Dr. WPF à l'adresse suivante :
**http://www.drwpf.com/blog/ItemsControlSeries/tabid/59/Default.aspx**
2 : L'implémentation originale de la classe RelayCommand provient également de son projet Crack.Net
3 : Une discussion très intéressante sur ce sujet est disponible sur l'archive des WPF Disciples :
**https://groups.google.com/forum/#!topic/wpf-disciples/P-JwzRB_GE8**