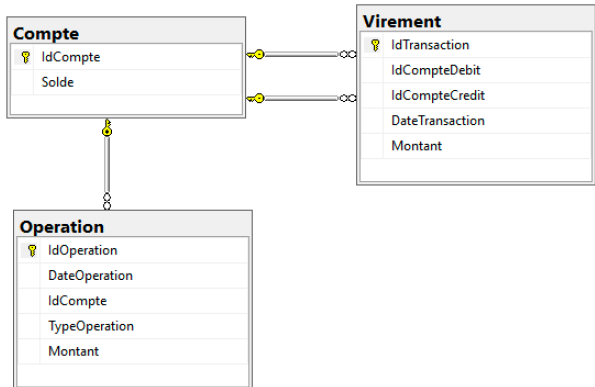


Nous allons refaire l'application de dépôt, retrait et virement bancaire du TD5 en intégrant 2 nouvelles bonnes pratiques de développement :

1. Utilisation d'un ORM (object-relational mapping) permettant l'accès aux données stockées dans une base SQL Server.
2. Application du pattern d'architecture MVVM (Model – Vue - Vue Modèle).

### 1. Création de la base de données

La base de données gère une liste de comptes, ainsi que des virements. Nous avons ajouté une table *Operation* permettant d'enregistrer toutes les opérations effectuées sur les comptes bancaires.

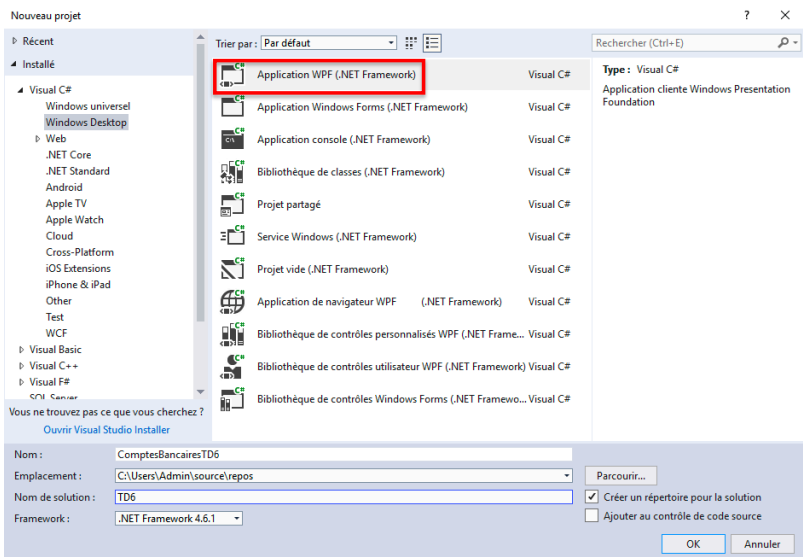


Dans votre base de données SQL Server, exécuter le script « Script ComptesBancaires SQL Server TD6.sql ». **Bien regarder le contenu du script SQL.**

### 2. Création du projet

Lancer Visual Studio 2017.

Créer un nouveau projet « Application WPF (.NET Framework) » :

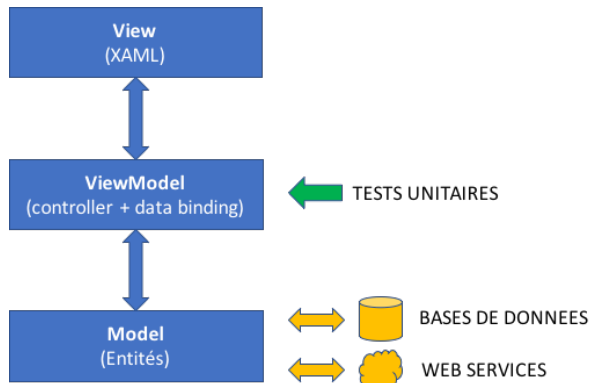


### 3. Codage de l'application en MVVM

#### 3.1. MVVM ?

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.



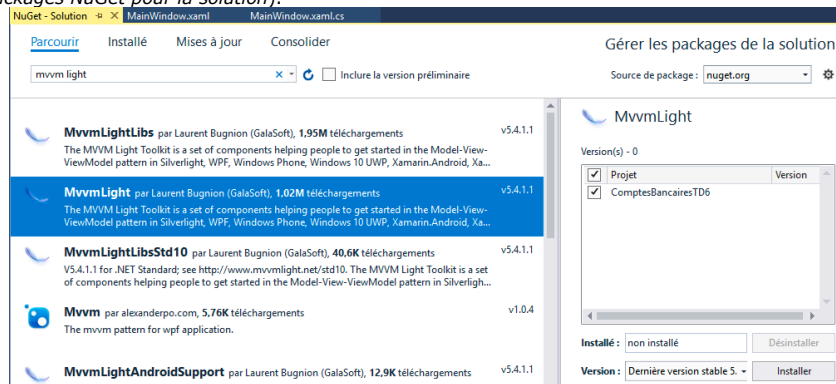
Le but de MVVM est de faire en sorte que la vue n'effectue aucun traitement : elle ne doit faire qu'afficher les données présentées par le ViewModel. C'est le ViewModel qui est chargé de réaliser les traitements et d'accéder au modèle. Les tests en seront facilités, car la vue ne contenant aucun code, 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 un framework nommé « MVVM Light ». MVVM Light va ainsi nous aider à mettre en place ce pattern. Il fournit notamment une architecture de projet compatible MVVM.

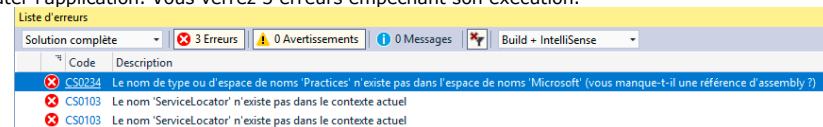
Lire cet excellent article sur MVVM Light : <http://blog.soat.fr/2015/06/mvvm-light-toolkit/>

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 !).

Ajouter le package NuGet « MvvmLight » au projet (Menu Outils > Gestionnaire de packages NuGet > Gérer les packages NuGet pour la solution).



Exécuter l'application. Vous verrez 3 erreurs empêchant son exécution.



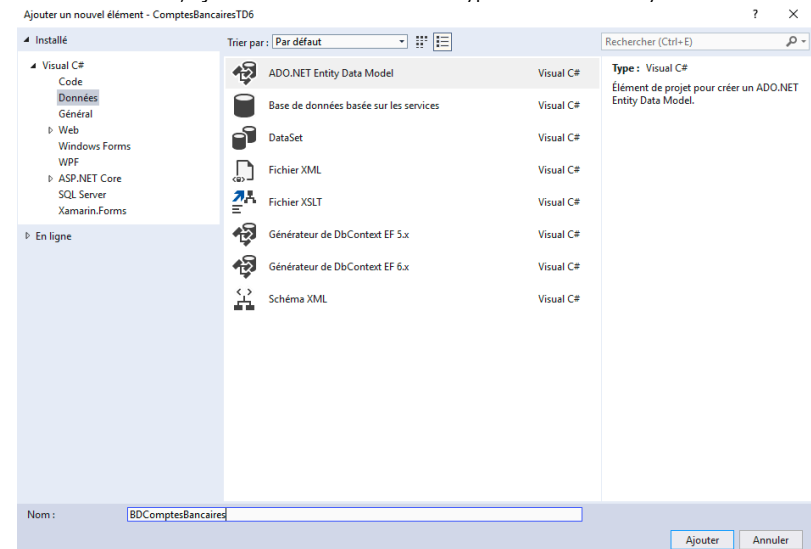
Cliquer sur la 1<sup>ère</sup> erreur. Remplacer `using Microsoft.Practices.ServiceLocation` par `using CommonServiceLocator`.

Vous ne devriez plus avoir d'erreur. Exécuter l'application.

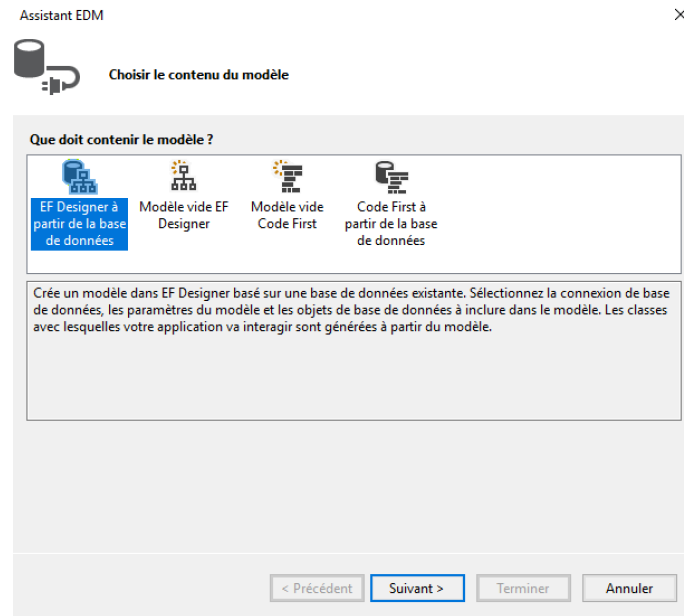
### 3.2. Couche Model

Créer un dossier Model et un sous-dossier Entity.

Dans le sous-dossier Entity, créer un modèle de données en utilisant Entity Framework, l'ORM de la plateforme .NET. Pour cela, ajouter un nouvel élément de type « ADO.NET Entity Data Model ».



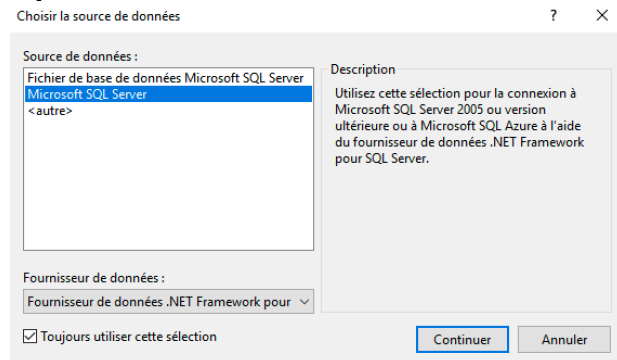
Choisir ensuite le type d'EF utilisé (DB-First, i.e. les classes métiers vont être générées à partir de la base de données => mapping relationnel vers objet).



Cliquer sur « Suivant ».

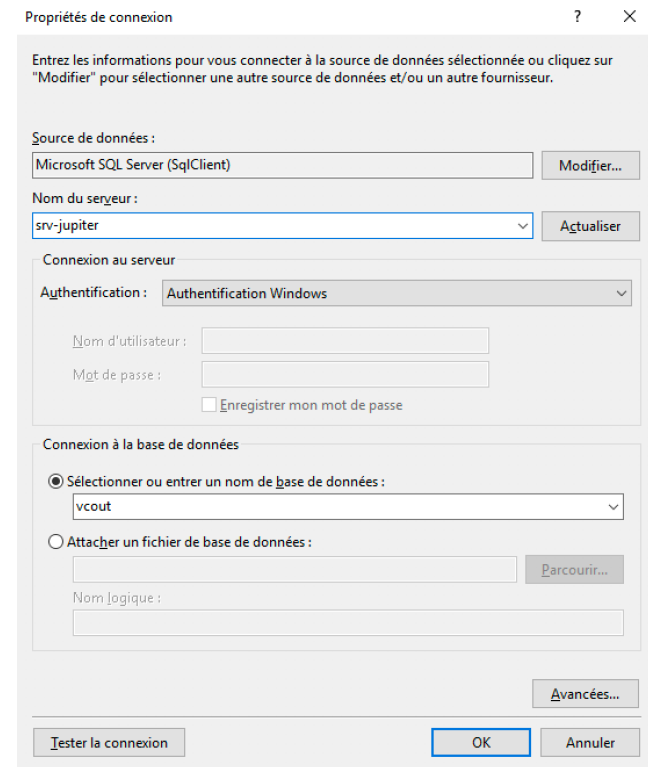
Cliquer ensuite sur le bouton « Nouvelle connexion ».

Choisir « Microsoft SQL Server ».



Entrer les informations suivantes :

**Bien sélectionner la base de données correspondant à votre login.**



Tester la connexion.

**Modifier le nom de la connexion.**

Assistant EDM

Choisir votre connexion de données

Quelle connexion de données votre application doit-elle utiliser pour établir une connexion à la base de données ?

desktop-p14qaea\sqlexpress.vcout.dbo Nouvelle connexion...

Cette chaîne de connexion semble contenir des données sensibles (par exemple, un mot de passe), lesquelles sont indispensables pour établir une connexion à la base de données. Le stockage des données sensibles dans la chaîne de connexion peut entraîner un risque de sécurité. Voulez-vous inclure les données sensibles dans la chaîne de connexion ?

☐ Non, exclure les données sensibles de la chaîne de connexion. Je définirai ces informations dans le code de mon application.

☐ Oui, inclure les données sensibles dans la chaîne de connexion.

Chaîne de connexion :

metadata=res://\*/Model.Entity.BDComptesBancaires.csdl|res://\*/Model.Entity.BDComptesBancaires.ssdl|res://\*/Model.Entity.BDComptesBancaires.msl;provider=System.Data.SqlClient;provider connection string="data source=localhost\SQLEXPRESS;initial catalog=vcout;integrated

☒ Enregistrer les paramètres de connexion dans App.Config en tant que :

BDComptesBancairesContext

< Précédent Suivant > Terminer Annuler

BDComptesBancairesContext correspond à la chaîne de connexion qui sera utilisée ultérieurement.

Choisir la version « Entity Framework 6.x ».

Sélectionner **les 3 tables** et modifier l'espace de nom du modèle (convention de nommage).

Remarque : Nous n'utiliserons pas de vues, car dans ce cas, il est obligatoire de créer des procédures stockées pour gérer les mises à jour (insert, update, delete), comme nous l'avons fait en fin de TD5. Il serait préférable de procéder ainsi, mais cela complexifierait davantage l'apprentissage.

Assistant EDM

Choisir vos paramètres et objets de base de données

Quels objets de base de données voulez-vous inclure dans votre modèle ?

☒ Tables

☒ dbo

☒ Compte

☒ Operation

☐ sysdiagrams

☒ Virement

☐ Vues

☐ Procédures et fonctions stockées

☐ Mettre au pluriel ou au singulier les noms d'objets générés

☒ Inclure les colonnes de clés étrangères dans le modèle

☐ Importer les fonctions et les procédures stockées sélectionnées dans le modèle d'entité

Espace de noms du modèle :

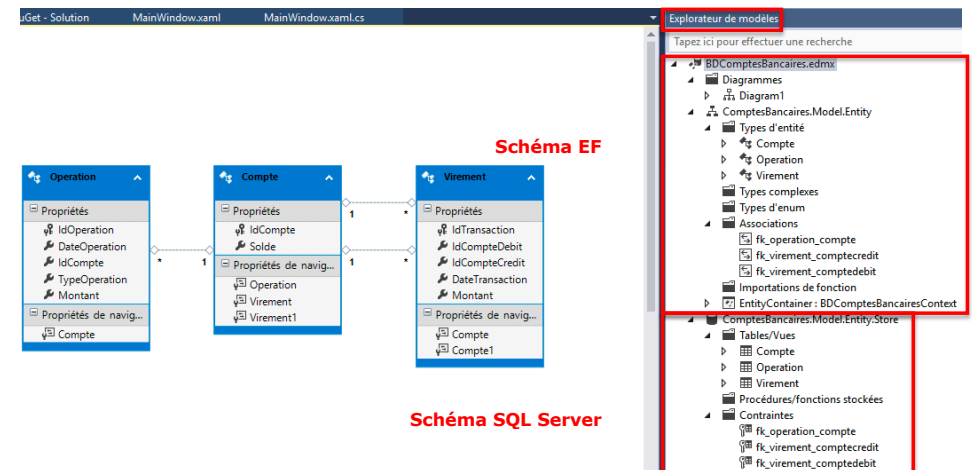
ComptesBancaires.Model.Entity

< Précédent Suivant > Terminer Annuler

Valider l'avertissement de sécurité.

### Schéma EF


Afficher la fenêtre « Explorateur EDM » (menu *Affichage* > *Autres fenêtres*).

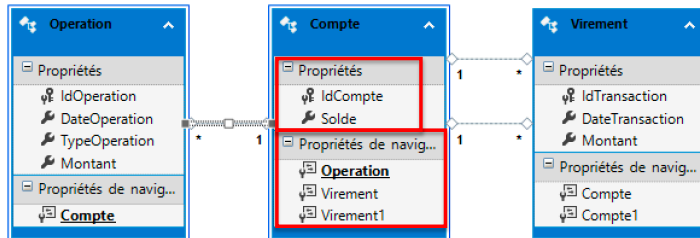


Un modèle .edmx a été créé. C'est un fichier XML qui définit un modèle conceptuel (objet), un modèle de stockage (relationnel) et le mapping entre ces modèles. Vous pouvez ci-dessus visualiser graphiquement le modèle conceptuel (modèle objet issu, dans notre cas, du mapping relationnel -> objet).

### Les propriétés

Les entités présentent deux types de propriété :

- Les propriétés standards qui sont le reflet de la base de données. Certaines propriétés sont définies comme PK .
- Les propriétés de navigation, qui permettent d'accéder aux objets associés (si plusieurs tables ou vues ont été sélectionnées).
  - Par exemple, la propriété `Compte` d'`Operation` permet d'accéder au compte figurant dans l'opération.
  - Inversement, la propriété `Operation` de `Compte` permet d'accéder à la liste des opérations effectuées sur ce compte.



Les propriétés standard sont associées à un mapping. Pour afficher le mapping faire un clic droit sur une entité puis *Mapping de table*. Dans notre cas :

Colonne	Opérateur	Valeur/Propriété
<b>Détails de mappage - Operation</b>		
<b>Tables</b>		
Est mappé à Operation		
<Ajouter une Condition>		
<b>Mappage de colonnes</b>		
IdOperation : int	↔	IdOperation : int32
DateOperation : datetime	↔	DateOperation : DateTime
IdCompte : int	↔	
TypeOperation : varchar	↔	TypeOperation : String
Montant : numeric	↔	Montant : Decimal
<Ajouter une table ou une vue>		

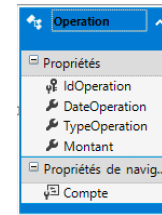
**Types SQL Server**

**Types .NET**

Les propriétés .NET (à droite) peuvent être renommées, modifiées et supprimées. Attention à cependant garder un mapping cohérent avec la base de données. Toutes les propriétés de l'entité doivent être mappées à une colonne de la base de données et toutes les colonnes NOT NULL de la base de données doivent se retrouver dans les entités.

A chaque propriété, sont également associées des caractéristiques (par exemple l'accessibilité, la taille max...). Ces caractéristiques sont modifiables dans la fenêtre des propriétés.

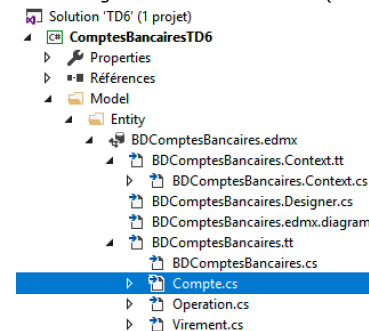
Par exemple, pour la property « TypeOperation » de l'entité « Operation » :



Propriétés	
ComptesBancaires.Model.Entity.Operation.TypeOperation Property	
<b>Facettes</b>	
Longueur fixe	False
Longueur max.	10
Unicode	False
<b>Général</b>	
Clé d'entité	False
<b>Documentation</b>	
Mode d'accès concurrentiel	None
Nom	TypeOperation
Nullable	False
StoreGeneratedPattern	None
Type	String
Valeur par défaut	(Aucune)
<b>Génération de code</b>	
Accesseur Get de propriété	Public
Accesseur Set de propriété	Public

### Les classes métier

Dans l'explorateur de solutions, regarder les classes métiers générées (dossier `Entity`). Le modèle graphique vu précédemment a généré 3 classes métiers (car 3 tables).



Les classes métier ne doivent pas être modifiées, car les modifications seraient perdues lors de la régénération du modèle.

Visualiser le code de la classe métier `Compte`.

```

namespace ComptesBancairesTD6.Model.Entity
{
    using System;
    using System.Collections.Generic;

    5 références
    public partial class Compte
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2214:DoNotCallOverridableMethodsInConstructors")]
        0 références
        public Compte()
        {
            this.Operation = new HashSet<Operation>();
            this.Virement = new HashSet<Virement>();
            this.Virement1 = new HashSet<Virement>();
        }

        0 références
        public int IdCompte { get; set; }
        0 références
        public decimal Solde { get; set; }

        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2227:CollectionPropertiesShouldBeReadOnly")]
        1 référence
        public virtual ICollection<Operation> Operation { get; set; }
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2227:CollectionPropertiesShouldBeReadOnly")]
        1 référence
        public virtual ICollection<Virement> Virement { get; set; }
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage", "CA2227:CollectionPropertiesShouldBeReadOnly")]
        1 référence
        public virtual ICollection<Virement> Virement1 { get; set; }
    }
}

```

On remarque qu'il s'agit d'une classe partielle (partial). Une classe partielle peut être complétée par une (ou plusieurs) autre classe partielle ayant le même nom. Le code des classes partielles sera assemblé à la compilation pour former un seul code. On retrouve les 3 propriétés de navigation qui sont des *ICollection* (par exemple, la *ICollection* *Operation* permet d'accéder à toutes les opérations du compte). Les propriétés de navigation sont toujours définies comme virtual.

Operation.cs :

```

namespace ComptesBancairesTD6.Model.Entity
{
    using System;
    using System.Collections.Generic;

    3 références
    public partial class Operation
    {
        0 références
        public int IdOperation { get; set; }
        0 références
        public System.DateTime DateOperation { get; set; }
        0 références
        public string TypeOperation { get; set; }
        0 références
        public decimal Montant { get; set; }

        0 références
        public virtual Compte Compte { get; set; }
    }
}

```

La property de navigation *Compte* permet d'accéder au compte de l'opération.

Virement.cs :

```

namespace ComptesBancairesTD6.Model.Entity
{
    using System;
    using System.Collections.Generic;

    5 références
    public partial class Virement
    {
        0 références
        public int IdTransaction { get; set; }
        0 références
        public System.DateTime DateTransaction { get; set; }
        0 références
        public decimal Montant { get; set; }

        0 références
        public virtual Compte Compte { get; set; }
        0 références
        public virtual Compte Compte1 { get; set; }
    }
}

```

*Compte* permet d'accéder au compte au compte de crédit ; *Compte1* au compte de débit. Nous allons renommer ces propriétés dans le modèle car leur nom est peu significatif.

Propriétés	
ComptesBancaires.Model.Entity.Virement.CompteCredit NavigationP	
Général	
Documentation	
Multiplicité	1 (un)
Nom	CompteCredit
Type de retour	Instance de Compte
Génération de code	
Accesseur Get de propriété	Public
Accesseur Set de propriété	Public
Navigation	
Association	fk_virement_comptecredit
Rôle cible	Compte
Rôle source	Virement

Idem pour la 2<sup>de</sup> property.

Le code de la classe est également modifié :

```

namespace ComptesBancairesTD6.Model.Entity
{
    using System;
    using System.Collections.Generic;

    5 références
    public partial class Virement
    {
        0 références
        public int IdTransaction { get; set; }
        0 références
        public int IdCompteDebit { get; set; }
        0 références
        public int IdCompteCredit { get; set; }
        0 références
        public System.DateTime DateTransaction { get; set; }
        0 références
        public decimal Montant { get; set; }

        0 références
        public virtual Compte CompteCredit { get; set; }
        0 références
        public virtual Compte CompteDebit { get; set; }
    }
}

```

La classe de contexte est contenue dans le fichier *BDComptesBancaires.Context.cs*.

```

namespace ComptesBancairesTD6.Model.Entity
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

    1 référence
    public partial class BDComptesBancairesContext : DbContext
    {
        Oréférences
        public BDComptesBancairesContext()
            : base("name=BDComptesBancairesContext")
        {
        }

        Oréférences
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            throw new UnintentionalCodeFirstException();
        }

        Oréférences
        public virtual DbSet<Compte> Compte { get; set; }
        Oréférences
        public virtual DbSet<Operation> Operation { get; set; }
        Oréférences
        public virtual DbSet<Virement> Virement { get; set; }
    }
}

```

La classe `BDComptesBancairesContext` représente le contexte de base de données Entity Framework et va s'occuper des opérations de création, lecture, mise à jour et suppression pour nous (CRUD). La classe de contexte créée hérite de la classe `DbContext` ([http://msdn.microsoft.com/fr-fr/library/system.data.entity.dbcontext\(v=vs.113\).aspx](http://msdn.microsoft.com/fr-fr/library/system.data.entity.dbcontext(v=vs.113).aspx)) et appelle le constructeur de cette dernière en lui passant le nom de la chaîne de connexion stockée dans le fichier `App.config`.

```

<connectionStrings>
  <add name="BDComptesBancairesContext" connectionString="metadata=res://*/Model.Entity.BDComptesBancaires.csd|res://*/Model.Entity.BDComptesBancairesContext.dll|provider=System.Data.SqlClient;providerAssembly=System.Data.SqlClient, Version=10.0.179.0, Culture=neutral, PublicKeyToken=b8951f41-90abf1a6" />
</connectionStrings>

```

On remarque la property `Compte` qui permet de manipuler les objets métiers `Compte` et notamment de les récupérer sous la forme d'une collection (`DbSet`). Idem pour les 2 autres properties.

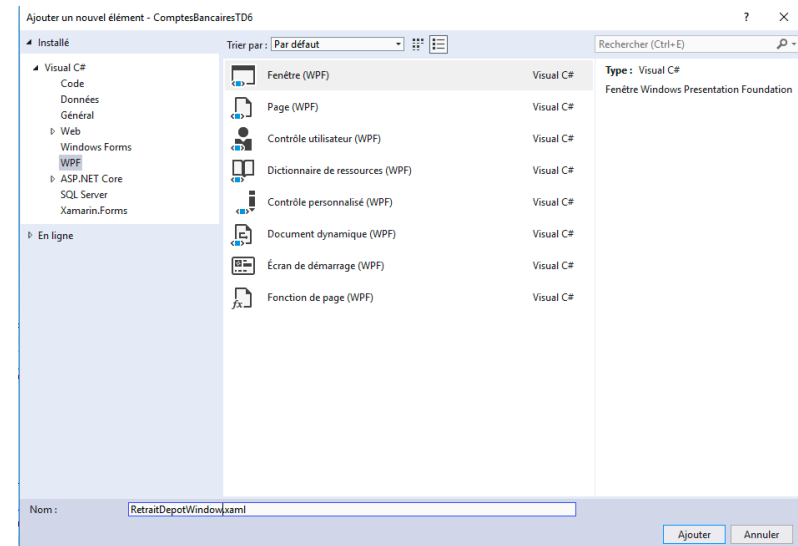
Notre couche *Model* est maintenant fonctionnelle.

## Générer la solution.

### 3.3. Couche View

Supprimer la fichier `MainWindow.xaml`.

Créer un dossier `View`. Ajouter une fenêtre `RetraitDepotWindow.xaml` dans ce dossier.



Vous pouvez modifier le titre de la fenêtre.

Code cs :

```

RetraitDepotWindow.xaml.cs  RetraitDepotViewModel.cs  RetraitDepotPage.xaml
ComptesBancairesTD6  ComptesBancairesTD6
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using System.Windows;
7  using System.Windows.Controls;
8  using System.Windows.Data;
9  using System.Windows.Documents;
10 using System.Windows.Input;
11 using System.Windows.Media;
12 using System.Windows.Media.Imaging;
13 using System.Windows.Shapes;
14
15 namespace ComptesBancairesTD6.View
16 {
17     /// <summary>
18     /// Logique d'interaction pour RetraitDepotWindow.xaml
19     /// </summary>
20     2 références
21     public partial class RetraitDepotWindow : Window
22     {
23         Oréférences
24         public RetraitDepotWindow()
25         {
26             InitializeComponent();
27         }
28     }
}

```

Nous n'ajouterons pas de code C# supplémentaire dans la vue. Tout se fera dans le `ViewModel`.

Dans le fichier `App.xaml`, modifier l'appel à la fenêtre `RetraitDepotWindow`.

```

<Application x:Class="ComptesBancairesTD6.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:ComptesBancairesTD6"
    StartupUri="View\RetraitDepotWindow.xaml"

```

```

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
dipl:Ignorable="d"
xmlns:dipl="http://schemas.openxmlformats.org/markup-compatibility/2006">
<Application.Resources>
  <ResourceDictionary>
    <vm:ViewModelLocator x:Key="Locator" d:IsDataSource="True" xmlns:vm="clr-
namespace:ComptesBancairesTD6.ViewModel" />
  </ResourceDictionary>
</Application.Resources>
</Application>

```

Exécuter l'application.

Réaliser l'interface suivante dans la vue XAML :

Opération à effectuer	
Compte	
Montant	
Valider	Annuler

Ce code XAML a été fait dans le TD5. **ATTENTION à supprimer toutes les actions Click si vous le récupérez.**

Exécuter l'application pour voir si tout fonctionne bien.

### 3.4. Création du ViewModel

Un dossier ViewModel a été ajouté suite à l'installation de MVVM Light.

Regarder la classe MainViewModel qui a été générée lors de l'installation de MVVM Light. Il s'agit du ViewModel de la fenêtre MainWindow.xaml. Elle hérite de ViewModelBase.

Nous n'avons pas besoin de cette classe ; la supprimer.

Ajouter la classe RetraitDepotViewModel.

RetraitDepotViewModel est le ViewModel correspondant à la vue RetraitDepotWindow. C'est cette classe qui coordonnera les échanges entre la vue et le modèle. Comme pour MainViewModel, elle doit hériter de la classe de base GalaSoft.MvvmLight.ViewModelBase fournie par MVVM Light.

```

1 using GalaSoft.MvvmLight;
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ComptesBancairesTD6.ViewModel
9 {
10     public class RetraitDepotViewModel : ViewModelBase
11     {
12     }
13 }
14

```

La classe ViewModelLocator a également été ajoutée par MVVM Light.

Dans un projet MVVM, la coordination entre la vue et le ViewModel est assurée par une classe spécifique nommée ViewModelLocator, puisque la vue n'a aucune connaissance du ViewModel (aucun code relatif à la vue n'est dans le ViewModel et inversement). C'est donc cette classe qui va permettre à la fenêtre (ou page) de trouver son ViewModel.

```

namespace ComptesBancairesTD6.ViewModel
{
    /// <summary>
    /// This class contains static references to all the view models in the
    /// application and provides an entry point for the bindings.
    /// </summary>
    public class ViewModelLocator
    {
        /// <summary>
        /// Initializes a new instance of the ViewModelLocator class.
        /// </summary>
        public ViewModelLocator()
        {
            ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);
            SimpleIoc.Default.Register<RetraitDepotViewModel>();
        }

        public RetraitDepotViewModel RetraitDepot
        {
            get
            {
                return ServiceLocator.Current.GetInstance<RetraitDepotViewModel>();
            }
        }

        public static void Cleanup()
        {
            // TODO Clear the ViewModels
        }
    }
}

```

SimpleIoc.Default.Register<RetraitDepotViewModel> permet d'enregistrer la classe RetraitDepotViewModel dans le "framework" MVVM Light ou plus exactement dans le conteneur d'inversion de contrôle (IoC). L'IoC est utilisée pour assigner les implémentations à leurs interfaces et les renvoyer à la demande. La méthode SimpleIoc.Default.Register permet ainsi d'assigner une classe à son interface.

Nous ne rentrerons pas plus dans le détail de ce qu'est l'inversion de contrôle, mais pour plus de détail, voir <http://blog.soat.fr/2015/06/mvvm-light-toolkit/>

La property public RetraitDepotViewModel RetraitDepot permet de récupérer l'occurrence de la classe RetraitDepotViewModel (à partir de son interface) correspondant à l'instanciation du ViewModel



(c'est la méthode `ServiceLocator.Current.GetInstance` qui renvoie un objet à partir de son interface).

### 3.5. 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 `RetraitDepotViewModel` le code suivant et positionner un point d'arrêt sur la ligne.

```
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ComptesBancairesTD6.ViewModel
9 {
10     4 références
11     public class RetraitDepotViewModel : ViewModelBase
12     {
13         0 références
14         public RetraitDepotViewModel()
15         {
16             Console.WriteLine("Je passe dans le ViewModel");
17         }
18     }
19 }
```

Exécuter ensuite l'application.

A aucun moment, le code n'est exécuté.

Rajouter maintenant le code en gras dans la vue xaml :

```
<Window x:Class="ComptesBancairesTD6.View.RetraitDepotWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:ComptesBancairesTD6.View"
mc:Ignorable="d"
Title="Opération bancaire" Height="450" Width="800"
DataContext="{Binding RetraitDepot, Source={StaticResource Locator}}">
<Grid>
...

```

Ce code permet de lier la vue à son ViewModel et utilise pour cela la property `RetraitDepot` que nous avons créée précédemment dans la classe `ViewModelLocator`. Cette property contient une instance de `RetraitDepotViewModel`.

Exécuter à nouveau l'application avec le point d'arrêt. Cette fois le code du constructeur est bien exécuté.

Locator est défini dans le fichier App.xaml :

```
<Application x:Class="ComptesBancairesTD6.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="clr-namespace:ComptesBancairesTD6"
StartupUri="View\RetraitDepotWindow.xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
d:lp1:Ignorable="d"
xmlns:d1p1="http://schemas.openxmlformats.org/markup-compatibility/2006">
<Application.Resources>
<ResourceDictionary>
<vm:ViewModelLocator x:Key="Locator" d:IsDataSource="True" xmlns:vm="clr-namespace:ComptesBancairesTD6.ViewModel" />
</ResourceDictionary>
</Application.Resources>
</Application>
```

Il s'agit d'une clé qui fait référence à la classe `ViewModelLocator` contenu dans le namespace `ComptesBancairesTD6.ViewModel`, autrement dit dans le dossier `ViewModel` de notre application. Ce code a été ajouté lors de l'installation du package NuGet.

### Remarque :

Si nous souhaitons ajouter une seconde fenêtre, il faut donc suivre les étapes suivantes :

1. Ajouter une fenêtre WPF xaml dans le dossier View. Elle sera, par exemple, nommée `Window2.xaml`.
2. Créer une classe ViewModel nommée par exemple `Window2ViewModel` dans le dossier `ViewModel`.

Cette classe doit héritée de la classe `ViewModelBase`.

using GalaSoft.MvvmLight;

```
namespace ConvertisseurWPF.ViewModel
{
    public class Window2ViewModel : ViewModelBase
    {
        public Window2ViewModel()
        {
        }
    }
}
```

3. Dans la classe `ViewModelLocator`, ajouter le code permettant d'enregistrer la classe `Window2ViewModel` dans le "framework" et de créer la property.

```
public class ViewModelLocator
{
    /// <summary>
    /// Initializes a new instance of the ViewModelLocator class.
    /// </summary>
    public ViewModelLocator()
    {
        ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);
        SimpleIoc.Default.Register<RetraitDepotViewModel>();
        SimpleIoc.Default.Register<Window2ViewModel>();
    }

    public RetraitDepotViewModel RetraitDepot
    {
        get
        {
            return ServiceLocator.Current.GetInstance<RetraitDepotViewModel>();
        }
    }

    public Window2ViewModel Window2
    {
        get
        {
            return ServiceLocator.Current.GetInstance<Window2ViewModel>();
        }
    }

    public static void Cleanup()
    {
    }
}
```

4. Ajouter le lien vers le ViewModel dans le fichier xaml de la vue.

```
<Window x:Class="ConvertisseurWPF.View.Window2"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:ConvertisseurWPF.View"
DataContext="{Binding Window2, Source={StaticResource Locator}}"
mc:Ignorable="d"
Title="2eme fenetre" Height="300" Width="300">
```

```
<Grid>
...
```

### 3.6. 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 pratiquement du même code que celui du TD5. **Seuls changements :**

- `OnPropertyChanged("MaProperty")` n'existe pas dans MVVM Light. Il faut utiliser `RaisePropertyChanged("MaProperty")` ou plus simplement `RaisePropertyChanged()`. Cette méthode permet de notifier la vue de tout changement dans le ViewModel.
- 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 *Codage de l'action sur le bouton « Valider »* plus loin pour les indications à suivre.

#### Codage du chargement des données dans la ComboBox « Compte » :

Dans le constructeur du ViewModel, ajouter le code suivant permettant de récupérer la liste des comptes de la base de données en utilisant EF :

```
private readonly BDComptesBancairesContext context;
public RetraitDepotViewModel()
{
    //Instanciation de la classe de contexte créée précédemment permettant d'accéder
    à la BD et de réaliser les opérations CRUD
    context = new BDComptesBancairesContext();
    // Exécution de context.Compte qui permet de retourner tous les comptes dans
    un DbSet<Compte>
    // Le DbSet<Compte> est ensuite transformé en ObservableCollection<Compte>
    this.... = new ObservableCollection<Compte>(context.Compte);
}
```

Lancer l'application. Normalement, les 2 listes devraient s'afficher.

Opération à effectuer	Retrait
Compte	1234567
Montant	0
Valider	Annuler

#### Codage de l'action sur le bouton « Valider » :

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 `ICommand` (<https://msdn.microsoft.com/fr-fr/library/system.windows.input.icommand%28v=vs.95%29.aspx>). Par exemple, le contrôle `Button` possède (par héritage) une propriété `Command` du type

Vincent COUTURIER

19

`ICommand` (<https://msdn.microsoft.com/fr-fr/library/system.windows.controls.primitives.buttonbase.command%28v=vs.95%29.aspx>) 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 (<http://blog.soat.fr/2015/06/mvvm-light-toolkit/>).

- Dans le fichier XAML, ajouter le code suivant :

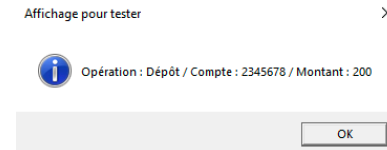
```
<Button Content="Valider" ... Command="{Binding BtnSetOperationBancaire}"/>
```

- Dans le fichier `MainViewModel`, ajouter le code suivant permettant de gérer le bouton :

```
public ICommand BtnSetOperationBancaire { get; private set; }
```

```
public RetraitDepotViewModel()
{
    ...
    BtnSetOperationBancaire = new RelayCommand(ActionSetOperationBancaire);
}
private void ActionSetOperationBancaire()
{
    //Code du calcul à écrire
}
```

Using à ajouter : `GalaSoft.MvvmLight.CommandWpf` et non `GalaSoft.MvvmLight.Command`;  
Le bouton est maintenant créé. Il appelle une méthode nommée `ActionSetOperationBancaire` à coder.  
Pour le moment, coder uniquement l'affichage des données :



La mise à jour du compte s'effectue de la façon suivante :

```
//On accède au compte de la BD correspondant au compte sélectionné via le contexte
//La restriction se fait via l'IdCompte
//Il s'agit d'une requête Linq écrite en utilisant des expressions Lambda
Compte cpt = context.Compte.First(c => c.IdCompte == CompteSelectionne.IdCompte);
if (this.OperationSelectionnee == "Retrait")
    cpt.Solde -= Convert.ToDecimal(this.Montant);
else if (this.OperationSelectionnee == "Dépôt")
    cpt.Solde += Convert.ToDecimal(this.Montant);
//On commite les changements sur le compte via le contexte de base de données
context.SaveChanges();
```

Requête Linq :

`context.Compte.First(c => c.IdCompte == CompteSelectionne.IdCompte)` est une requête Linq. C'est une syntaxe qui semble au départ assez peu « naturelle » mais qui permettra une fois maîtrisée de simplifier grandement la gestion des listes (car Linq ne s'applique pas qu'à Entity Framework, mais à n'importe quelle collection) : <https://msdn.microsoft.com/en-us/library/bb397678.aspx>

Il y a 2 manières d'écrire une requête LINQ :

- En lambda expression (`c => c...`). Ce sont en fait des fonctions anonymes :
  - o `context.Compte.First(c => c.IdCompte == CompteSelectionne.IdCompte)`  
La méthode `First` (<http://www.entityframeworktutorial.net/queries/entity-framework.aspx>) renvoie le premier `Compte` trouvé validant l'expression `c => c.IdCompte == CompteSelectionne.IdCompte`. Le compte est renvoyé dans `c`.
- En pseudo SQL, la requête précédente s'écrit :  

```
Compte cpt =
    (from c in context.Compte
     where c.IdCompte == CompteSelectionne.IdCompte
     select c).First();
```

#### Codage de l'action sur le bouton « Annuler » :

Vincent COUTURIER

20

L'opération sera remise à "" et le montant à 0.

On peut aussi sélectionner le 1<sup>er</sup> compte de la liste par défaut.

Penser à utiliser `RaisePropertyChanged()` dans les propriétés pour notifier la vue.

#### Gestion d'erreurs :

- Rajouter des blocs try/catch dans le constructeur, ainsi que dans chaque méthode.
- Il est possible de gérer une exception si le compte et/ou l'opération à effectuer n'est pas sélectionné. Il est cependant préférable de bloquer le bouton dans ces cas. Plus exactement, vous allez bloquer la commande associée au bouton. Pour cela, le code du `RelayCommand` devient :

```
BtnSetOperationBancaire = new RelayCommand(ActionSetOperationBancaire,
CanExecuteOperationBancaire);
```

```
RelayCommand.RelayCommand(Action execute, Func<bool> canExecute) (+ 1 surcharge)
Initializes a new instance of the RelayCommand class.

Exceptions:
ArgumentNullException
```

Le second paramètre correspond à une méthode retournant un booléen qu'il faut coder.

```
private bool CanExecuteOperationBancaire()
{
    return (this.CompteSelectionne!=null) &&
    !(String.IsNullOrEmpty(this.OperationSelectionnee));
}
```

#### 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.

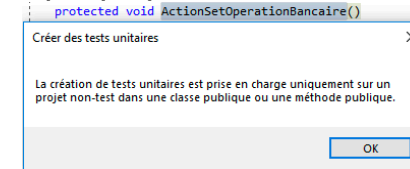
Dans notre cas, nous avons pleinement appliqué le pattern MVVM car aucun code ne figure dans le fichier `RetraitDepotWindow.xaml.cs`. Peu importe si vous ne respectez pas parfaitement le MVVM, le principe de ce pattern est de vous aider dans la réalisation de votre application et surtout dans sa maintenabilité.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

#### 4. Tests unitaires de la couche ViewModel

Il n'est pas nécessaire de tester la couche *Model* car générée par Entity Framework.

Hormis des tests d'IHM qui peuvent être réalisés sur la vue pour tester la navigation (si celle-ci n'a pas été codée en MVVM), la seule couche à tester est donc le ViewModel. Pour cela, nous allons réaliser des tests unitaires.

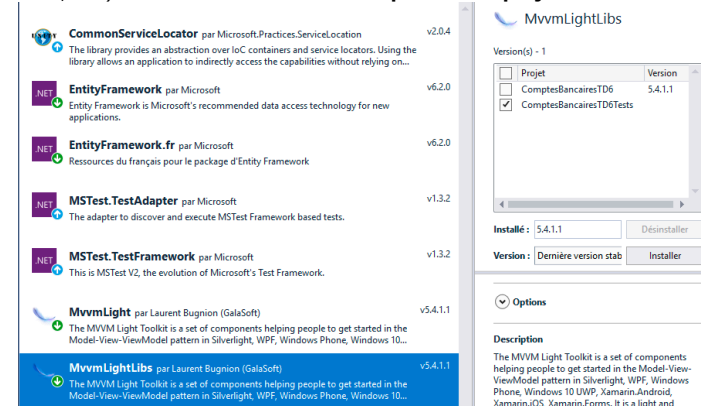
**Rappel : seules les méthodes publiques peuvent être testées. Il faut donc les définir en public.**



Cliquer sur une des méthodes publiques de `RetraitDepotViewModel` avec le bouton droit de la souris, puis choisir « Créer des tests unitaires » pour générer le projet de test `MSTestV2`.

Dans le projet de test, il est nécessaire d'installer les packages NuGet :

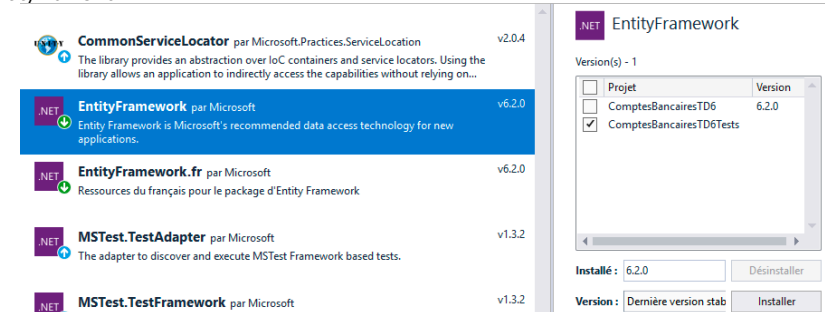
- `MvvmLightLibs` (seulement les librairies afin de ne pas installer les classes `ViewModelLocator`, `MainViewModel`, etc.). **Installer la même version que dans le projet WPF.**



Ces librairies sont nécessaires pour pouvoir notamment tester la méthode `ActionSetOperationBancaire`. Sinon, vous aurez l'erreur suivante :

Le type 'ViewModelBase' est défini dans un assembly qui n'est pas référencé. Vous devez ajouter une référence à l'assembly 'GalaSoft.MvvmLight, Version=5.4.1.0, Culture=neutral, PublicKeyToken=e7570ab207bcb616'.

- EntityFramework :



Le projet à tester utilisant EF, il est nécessaire de l'installer dans le projet de test. Sinon, vous aurez l'erreur suivante lors de l'exécution des tests :

```
ConstructorRetraitDepotViewModel Copier to
Source: RetraitDepotViewModelTests.cs ligne 15
✖ Test Échec - ConstructorRetraitDepotViewModel
Message : La méthode de test ComptesBancairesTD6.ViewModel.Tests.Retr
itDepotViewModel.ITests.ConstructorRetraitDepotViewModel a levé une exception :
System.InvalidOperationException: La chaîne de connexion 'BDComptesBancairesContext' est introuvable dans le fichier de configuration de l'application.
Temps écoulé : 0:00:01,2743891
```

On voit dans l'erreur précédente, qu'il faut aussi rajouter votre chaîne de connexion (BDComptesBancairesContext) dans le fichier App.config du projet de test (copier-coller celle de l'App.config du projet WPF).

```
<connectionStrings>
  <add name="BDComptesBancairesContext" connectionString="metadata=res://*/Model.Entity.BDComptesBancaires.csd|res://*/Model.Entity.BDComptesBancaires.csd|res://*/Model.Entity.BDComptesBancaires.csd|res://*/Model.Entity.BDComptesBancaires.csd" provider="System.Data.SqlClient" providerInvariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer, System.Data.Entity.SqlServer, System.Data.Entity, System.Data, System" />
</connectionStrings>
```

Résultat :

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <configuration>
3   <configSections>
4     <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?linkid=237468 -->
5     <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework" />
6   </configSections>
7   <entityFramework>
8     <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnectionFactory, EntityFramework" />
9     <providers>
10      <provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer, System.Data.Entity.SqlServer, System.Data.Entity, System.Data, System" />
11    </providers>
12  </entityFramework>
13  <connectionStrings>
14    <add name="BDComptesBancairesContext" connectionString="metadata=res://*/Model.Entity.BDComptesBancaires.csd|res://*/Model.Entity.BDComptesBancaires.csd|res://*/Model.Entity.BDComptesBancaires.csd|res://*/Model.Entity.BDComptesBancaires.csd" provider="System.Data.SqlClient" providerInvariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer, System.Data.Entity.SqlServer, System.Data.Entity, System.Data, System" />
15  </connectionStrings>
16 </configuration>
```

Méthodes de test à ajouter (et à lire) pour tester le retrait/dépôt (**A MODIFIER EN FONCTION DE VOTRE CODE**) :

- Test du constructeur :

```
[TestMethod()]
public void ConstructorRetraitDepotViewModelTest()
{
    RetraitDepotViewModel retraitDepot = new RetraitDepotViewModel();
    Assert.IsNotNull(retraitDepot);
}
```
- Test du bouton BtnSetOperationBancaire :  
Les méthodes BtnSetOperationBancaire\_CanExecute permettent de tester si le bouton est "activé" ou non.

```
[TestMethod()]
public void BtnSetOperationBancaireTest_CanExecuteFailure_NoData()
{
    RetraitDepotViewModel retraitDepot = new RetraitDepotViewModel();
    Assert.IsFalse(retraitDepot.BtnSetOperationBancaire.CanExecute(null));
}

[TestMethod()]
public void BtnSetOperationBancaireTest_CanExecuteFailure_OperationOnly()
{
    RetraitDepotViewModel retraitDepot = new RetraitDepotViewModel();
    retraitDepot.OperationSelectionnee="Retrait";
    Assert.IsFalse(retraitDepot.BtnSetOperationBancaire.CanExecute(null));
}

[TestMethod()]
public void BtnSetOperationBancaireTest_CanExecuteFailure_CompteOnly()
{
    var retraitDepot = new RetraitDepotViewModel();
    Compte cpt = new Compte
```

```
{
    IdCompte = 1234567,
    Solde = 1000
};
retraitDepot.CompteSelectionnee = cpt;
Assert.IsFalse(retraitDepot.BtnSetOperationBancaire.CanExecute(null));
}

[TestMethod()]
public void BtnSetOperationBancaireTest_CanExecuteSuccess()
{
    RetraitDepotViewModel retraitDepot = new RetraitDepotViewModel();
    Compte cpt = new Compte
    {
        IdCompte = 1234567,
        Solde = 1000
    };
    retraitDepot.CompteSelectionnee = cpt;
    retraitDepot.OperationSelectionnee="Retrait";
    Assert.IsTrue(retraitDepot.BtnSetOperationBancaire.CanExecute(null));
}
```

- Test de la méthode ActionSetOperationBancaire :

```
[TestMethod()]
public void ActionSetOperationBancaireTest_RetraitSuccess()
{
    //Arrange
    RetraitDepotViewModel retraitDepot = new RetraitDepotViewModel();
    // On récupère le compte 1234567 en utilisant une requête LINQ, comme on l'a fait dans la méthode ActionSetOperationBancaire
    BDComptesBancairesContext context = new BDComptesBancairesContext();
    Compte compteRetrait = context.Compte.First(c => c.IdCompte == 1234567);
    retraitDepot.CompteSelectionnee = compteRetrait;
    retraitDepot.OperationSelectionnee = "Retrait";
    retraitDepot.MontantOperation = 200;

    //Act
    retraitDepot.ActionSetOperationBancaire();
    //On récupère à nouveau le compte après mise à jour du solde dans la BD
    // On est obligé de créer un nouveau contexte, car sinon on aura toujours accès à l'ancien solde.
    BDComptesBancairesContext context2 = new BDComptesBancairesContext();
    Compte compteRecupere = context2.Compte.First(c => c.IdCompte == compteRetrait.IdCompte);

    //Assert
    //On vérifie le solde que l'on devrait avoir (solde du compte avant retrait - montant retrait) avec le solde dans la BD
    Assert.AreEqual(compteRetrait.Solde - Convert.ToDecimal(retraitDepot.MontantOperation), compteRecupere.Solde);
}

[TestMethod()]
public void ActionSetOperationBancaireTest_DepotSuccess()
{
    // A compléter
}
```

Remarque : à la place de la création du context2, on pourrait détruire le premier contexte puis le recréer :

```
context.Dispose();
BDComptesBancairesContext context = new BDComptesBancairesContext();
```

- Test de la méthode ActionCancel : le code devrait être plus simple...

Exécuter les tests.

Si vous lancez l'analyse de la couverture du code, vous verrez que le code de la classe `RetraitDepotViewModel` est pratiquement testé à 100% :

Résultats de la couverture du code				
Admin_DESKTOP-P14QAEA 2018-10-22 09_50_30.coverage				
Hierarchie	Non couverts (blocs)	Non couverts (% blocs)	Couverts (blocs)	Couverts (% blocs)
Admin_DESKTOP-P14QAEA 2018-10-22 09_50_30.coverage	42	14,29 %	252	85,71 %
comptesbancairestd6.exe	42	24,14 %	132	75,86 %
ComptesBancairesTD6.Model.Entity	17	31,48 %	37	68,52 %
ComptesBancairesTD6.Properties	5	100,00 %	0	0,00 %
ComptesBancairesTD6.View	3	100,00 %	0	0,00 %
ComptesBancairesTD6.ViewModel	17	15,18 %	95	84,82 %
RetraitDepotViewModel	2	2,06 %	95	97,94 %
ActionCancel()	0	0,00 %	3	100,00 %
ActionSetOperationBancaire()	0	0,00 %	47	100,00 %
CanExecuteOperationBancaire()	0	0,00 %	8	100,00 %
RetraitDepotViewModel()	0	0,00 %	21	100,00 %
get_BtnCancel()	1	100,00 %	0	0,00 %
get_BtnSetOperationBancaire()	0	0,00 %	1	100,00 %
get_CompteSelectionne()	0	0,00 %	1	100,00 %
get_Comptes()	1	100,00 %	0	0,00 %
get_MontantOperation()	0	0,00 %	2	100,00 %
get_OperationSelectionnee()	0	0,00 %	2	100,00 %
get_Operations()	0	0,00 %	1	100,00 %
set_BtnCancel(System.Windows.Input.ICom...	0	0,00 %	1	100,00 %
set_BtnSetOperationBancaire(System.Windo...	0	0,00 %	1	100,00 %
set_CompteSelectionne(ComptesBancairesT...	0	0,00 %	1	100,00 %
set_Comptes(System.Collections.ObjectMod...	0	0,00 %	1	100,00 %
set_MontantOperation(double)	0	0,00 %	2	100,00 %
set_OperationSelectionnee(string)	0	0,00 %	2	100,00 %
set_Operations(System.Collections.ObjectMo...	0	0,00 %	1	100,00 %
ViewModelLocator	13	100,00 %	0	0,00 %
ViewModelLocator.<.>c	2	100,00 %	0	0,00 %
comptesbancairestd6tests.dll	0	0,00 %	120	100,00 %

On voit également que le modèle EF est en grande partie testé, alors que nous n'avons écrit aucun test (des tests sont incorporés dans le framework).

Enfin, le `ViewModelLocator` n'est pas du tout testé, mais on ne le fait jamais...

## 5. Modifications

### 5.1. Ajout d'une nouvelle fenêtre

Dans le dossier `View`, ajouter une nouvelle fenêtre permettant de réaliser un virement. Modifier le code du `ViewModelLocator` pour prendre en charge cette seconde fenêtre (suivre la procédure de la page 18).

Pour ajouter un virement en utilisant le contexte de la base de données :

```
Virement virement = new Virement();
virement.xxx=...;
virement.zzz=...;
context.Virement.Add(virement);
context.SaveChanges();
```

Tester la fenêtre après avoir modifié le `StartupUri` dans le fichier `App.xaml`.

ATTENTION à bien utiliser le namespace `GalaSoft.MvvmLight.CommandWpf` et non `GalaSoft.MvvmLight.Command`, sinon vous aurez des soucis avec `CanExecute`.

Remarques :

Vincent COUTURIER

- Nous aurions pu gérer des pages plutôt que des fenêtres.
- Si vous souhaitez ajouter du code de navigation entre fenêtres et/ou pages, il est plus simple de le faire dans le `code-behind` (même si on peut aussi le faire en MVVM). En effet, en général, on applique le pattern MVVM sur le code fonctionnel, mais assez rarement sur la navigation. Il suffira ensuite de créer un test d'UI pour vérifier que la navigation entre les pages/fenêtres est bien conforme.

### Réaliser les tests unitaires du virement.

Pour le test de l'action liée au bouton « Valider », comme il n'y a pas de méthode `Equals` dans les classes métier (si vous en rajoutez une, elle sera supprimée lorsque vous re-générerez le modèle ou le mettez à jour), vous devrez tester chaque attribut du virement. Vous ferez donc plusieurs `Assert.AreEqual`.

### 5.2. Divers (pour les plus rapides)

- Enregistrement des opérations bancaires :  
A chaque retrait ou dépôt bancaire, une ligne dans la table `Operation` est ajoutée. `TypeOperation = {Dépôt, Retrait}` (attention à la contrainte check dans la base de données).  
Date/heure actuelle : `DateTime.Now`.  
A chaque virement, deux lignes (1 pour le compte crédité, 1 pour le compte débité) dans la table `Operation` sont ajoutées (à la même date/heure !).

**Modifier les tests unitaires.** Afin de simplifier, on vérifiera juste si le nombre de lignes prévu a bien été ajouté dans la table `Operation` (utiliser la méthode `count()` sur le `DBSet`).

- Améliorer/finaliser le fonctionnement de l'application.

25

Vincent COUTURIER

26