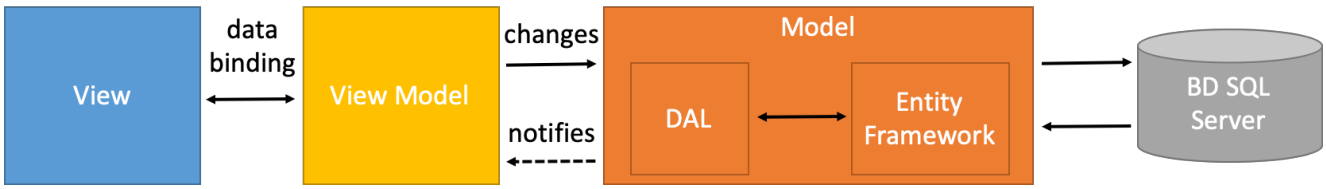
M3105 – Conception et programmation objet avancée

TD7 – Amélioration du code de la couche Model

Ce TD fait suite au TD6.

Pour éviter d'accéder directement aux données à partir des view models, nous allons ajouter une couche d'accès aux données, nommée *Data Access Layer (DAL)*. Cette couche va permettre d'assurer un couplage léger entre les View Model et les données (générées par Entity Framework), ce qui permettra de modifier la couche des données sans modifier celle des View Models (par exemple, si l'on souhaite utiliser un autre ORM comme Dapper, se passer d'ORM ou utiliser un autre moteur/format de stockage).



1. Création de la couche d'accès aux données

La couche d'accès aux données (*Data Access Layer*), que nous allons créer a pour but de faciliter l'accès aux données, de créer un point d'entrée unique et éventuellement de changer le type d'accès aux données sans perturber le reste du programme.
Le point d'entrée sera la classe `BDComptesBancairesContext` qui s'occupe des opérations de création, lecture, mise à jour et suppression (CRUD).

1. Créer un dossier `DAL` dans le dossier `Model`.

2. Définir le contrat du Data Access Layer dans le répertoire `Model/DAL`, appelé `IDal` : interface contenant les différentes méthodes dont nous allons avoir besoin. Cette interface dérive de l'interface `IDisposable` (qui définit une méthode `Dispose()` pour libérer des ressources allouées). Le code de l'interface est le suivant :

```
public interface IDal : IDisposable
{
    #region Opérations du compte
        bool AjouterCompte(int idCompte, decimal solde);
        bool ModifierSolde(int idCompte, decimal montant);
        Compte RenvoieCompte(int idCompte);
        IQueryable<Compte> RenvoieTousLesComptes(); // ou List ou IList si vous
préférez
        bool CompteExists(int idCompte);
        // Pas de suppression
    #endregion

    #region Opérations du Virement
        // A compléter
    #endregion

    #region Opérations des Opérations bancaires
        // A compléter
    #endregion
}
```

`IQueryable` est plus performant quand de nombreuses données sont récupérées : https://www.codeproject.com/Articles/832189/List-vs-IEnumerable-vs-IQueryable-vs-ICollection-v. Vous pourrez utiliser une liste à la place.

3. Créer une classe `Dal` (dans le dossier `DAL`) qui implémente l'interface `IDal`. Cette classe est composée :
   a) D'un constructeur qui instancie une variable privée de la classe `BDComptesBancairesContext()`. Cela permet d'accéder aux différents éléments de la base de données.
   b) D'une méthode `Dispose()` qui utilise la méthode `Dispose()` de la classe `BDComptesBancairesContext`. Cette méthode permet de supprimer notre objet `BDComptesBancairesContext` dès que nous avons fini tous les traitements.
   c) Dans le contrat `IDal`, lever une exception `NotImplementedException()` dans chaque méthode non encore codée.

4. Créer le code des méthodes de la classe `Dal`. Exemple pour l'ajout d'un compte :

```
public bool AjouterCompte(int idCompte, decimal solde)
{
   if (CompteExists(idCompte))
         return false;
      else
      {
         Compte compte = new Compte()
         {
               IdCompte = idCompte,
               Solde = solde,
         };
         _context.Compte.Add(compte);    //    _context :    objet    de    type
BDComptesBancairesContext
         _context.SaveChanges();
         return true;
      }
}

public bool CompteExists(int idCompte)
{
   return _context.Compte.Count(c => c.IdCompte == idCompte) > 0;
}
```

Récupération de tous les comptes bancaires :

```
public IQueryable<Compte> RenvoieTousLesComptes()
{
      return _context.Compte;
}
```

2. ***Modification du code du View Model***

Modifier les méthodes du View Model de façon qu'elles utilisent la couche `Dal`.
Lancer l'application.

Vérifier que les tests sont toujours fonctionnels. Pour cela, vous devrez ajouter la méthode `AjouterOperation` (prototype : `bool AjouterOperation(int idCompte, string typeOperation, decimal montant)`) dans l'interface `IDal` et la coder. Appeler la méthode dans le View Model de façon à créer l'opération bancaire.

Remarque : dans un DAL, on ne créé en général que des paramètres de méthodes de type simple (type scalaire) et donc pas d'instance de `Compte` ou autre.

3. ***Dependency Injection***

Dans notre View Model, nous avons dû utiliser un objet `Dal`.

```
private readonly Dal dal;

public RetraitDepotViewModel()
{
    this.dal = new Dal();
}
```

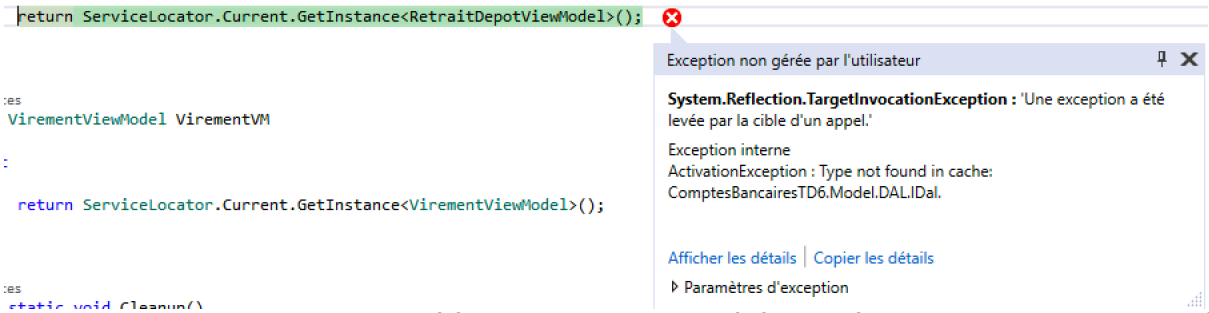
Cependant, le code précédent n'est pas très propre car nous avons lié notre contrôleur à une implémentation spécifique de `Dal`.
Dans notre application, lorsque l'on compile la couche View Model, elle référence (`new Dal()`) et utilise la couche d'accès aux données. Ainsi, si l'on doit changer de couche d'accès aux données, il faudra que l'on recompile l'application.
Le découplage va se baser sur l'interface que nous avons préalablement créée. La couche DAL publie ainsi ce qu'elle est capable de faire en implémentant l'interface. Le View Model possèdera une référence vers quelque chose qui implémente l'interface (il ne sait pas forcément qui) mais ne s'en préoccupera pas. Ainsi, il n'y aura plus de référence directe à la classe concrète implémentant le DAL.

Si la couche View Model ne se préoccupe plus de la classe concrète, qui instancie alors cette dernière ?
Pour cela, nous allons utiliser un mécanisme, appelé *dependency injection* (ou injection de dépendances), c'est-à-dire que nous allons travailler uniquement avec des interfaces et laisser un outil d'injection de dépendances, dans notre cas *Unity*, instancier la bonne implémentation du DAL.

L'injection de dépendances est un design pattern incontournable lorsque l'on souhaite développer une application à la fois volumineuse et modulaire. Avec ce design pattern, les composants n'ont pas besoin de connaitre la manière dont sont créées leurs dépendances. Ce pattern est utilisé dans beaucoup de frameworks de développement récents tels qu'Angular avec le décorateur `@Injectable()` (https://angular.io/api/core/Injectable), symfony, Android (https://www.raywenderlich.com/146804/dependency-injection-dagger-2), etc.

1. Modifier le code du View Model de façon à ce qu'il utilise l'interface. Le constructeur devient :
```
private readonly IDal dal;
public RetraitDepotViewModel(IDal dal)
{
    …
    this.dal = dal;
    …
}
```

2. Lancer l'application. Vous obtiendrez l'erreur suivante :



L'erreur indique qu'aucun type (classe) lié à l'interface IDal n'a été trouvé dans le cache. C'est le mécanisme d'injection de dépendance qui va avoir la charge de fournir ce type.

3. MVVM Light contient un mécanisme d'injection de dépendance de poids léger, extensible et aidant à la construction d'architectures découplées. Ce mécanisme est intégré au container d'IoC, disponible via la classe `SimpleIoc`. Pour enregistrer un objet concret lié à une interface (rajouter la ligne dans le `ViewModelLocator`) : `SimpleIoc.Default.Register<`*`IMonInterface`*`,` *`MaClasseConcrete`*`>()`. Ainsi, `SimpleIoc` enregistre l'interface et la classe qui l'implémente dans le cache. Plus de détails ici : https://msdn.microsoft.com/en-us/magazine/jj991965.aspx

En programmation orientée objet, de façon classique, un objet (classe ou module) contient un ensemble de dépendances envers d'autres objets, auxquels il va déporter tout ou partie de ses traitements. Le bon côté de la chose est que l'on évite ainsi que les objets contiennent trop de comportements (les rendant difficiles à maintenir). Le mauvais côté est que chacun de ces objets référencés devient une dépendance forte, car l'objet appelant doit connaitre chacun des objets qu'il va utiliser avant de les instancier.
Inversion de contrôle *: l'inversion de contrôle (IoC) est un concept de haut niveau en développement orienté objet. Ce concept veut que lorsqu'un module effectue un traitement, le contrôle du traitement soit déporté vers l'appelé, et non pas vers l'appelant. En pratique, on va chercher à diminuer au maximum*

la connaissance qu'a l'appelant de la mécanique interne de l'appelé. C'est ce mécanisme que nous avons utilisé quand nous avons écrit `SimpleIoc.Default.Register<MonViewModel>()` *puis* `ServiceLocator.Current.GetInstance<MonViewModel>()` *dans le* `ViewModelLocator`*. La vue (l'appelant) n'a ainsi pas connaissance du View Model (l'appelé). C'est le container d'IoC (*`SimpleIoC`*) qui va lui renvoyer le bon View Model (méthode* `GetInstance`*) stocké dans son cache (méthode* `Register`*).*
***Inversion de dépendances** : l'inversion de dépendances (DI) est un principe de développement orienté objet. Il s'agit d'une des implémentations possibles de l'inversion de contrôle. La définition simplifiée de ce principe est que, pour diminuer le couplage entre les classes, on va ajouter une interface entre chaque classe, de façon à ce qu'au lieu d'appeler une classe physique, l'appelant appelle une interface. Ceci permet d'ajouter un niveau d'abstraction supplémentaire entre l'appelant et l'appelé.*
***Injection de dépendances** (DI) : Technique permettant de mettre en œuvre l'inversion de dépendances. La résolution de la dépendance est déportée en dehors du code "métier" : on va, dans un premier temps, définir un jeu d'interfaces de façon à ce que nos différents modules puissent communiquer par un contrat ; dans un second temps, on va "injecter" dans notre objet un autre objet répondant au contrat défini. L'injection de dépendance de la couche Dal est réalisé par* `SimpleIoc.Default.Register<IDal, Dal>()`.

Utiliser l'inversion de contrôle offre les avantages suivants :

- *Chaque système ne se concentre que sur sa ou ses tâches principales.*
- *Les différents systèmes ne font pas d'hypothèse sur le comportement des autres systèmes.*
- *Remplacer un système ne produit pas d'effets de bord sur les autres systèmes, tant que le contrat d'origine est respecté.*
- *Dans le cas d'une nouvelle version d'un composant, il est plus facile de changer le composant appelé.*

Remarque : Microsoft fournit son propre container IoC, nommé Unity Application Block. Il intègre naturellement la résolution (injection) et la configuration des dépendances sous forme de génériques. C'est un outil suggéré par Microsoft pour un développement plus propre qui fait partie des Patterns & Practices de Microsoft (https://github.com/mspnp). Plutôt que d'installer un package NuGet supplémentaire, nous avons utilisé celui fourni par MVVM Light.

Exécuter l'application.

4. Les tests ne sont plus fonctionnels.

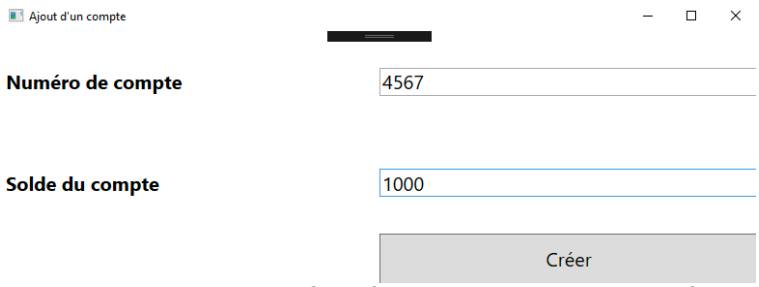
Ajouter le code suivant :

```
private IDal dal;

public RetraitDepotViewModelTests() // Constructeur des tests
{
    dal = new Dal();
}

public void Dispose() // Pour libérer le DAL à la fin des tests
{
    dal.Dispose();
}
```

Les appels au View Model deviennent : `RetraitDepotViewModel retraitDepot = new RetraitDepotViewModel(dal);`

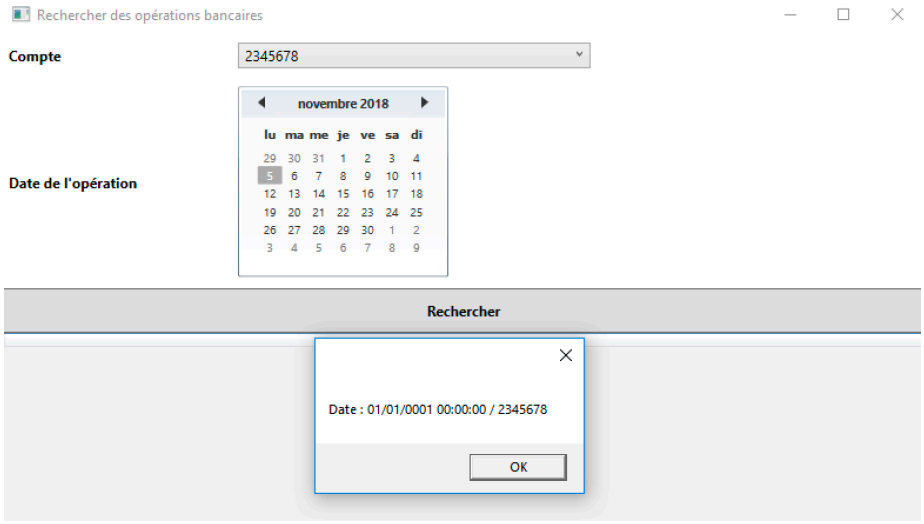
4. ***Ajouts fonctionnels***

- Compléter le DAL pour gérer les virements. Modifier en conséquence le View Model, ainsi que les tests.
- Ajouter une fenêtre (ou une page) permettant d'ajouter un compte bancaire. Par exemple :

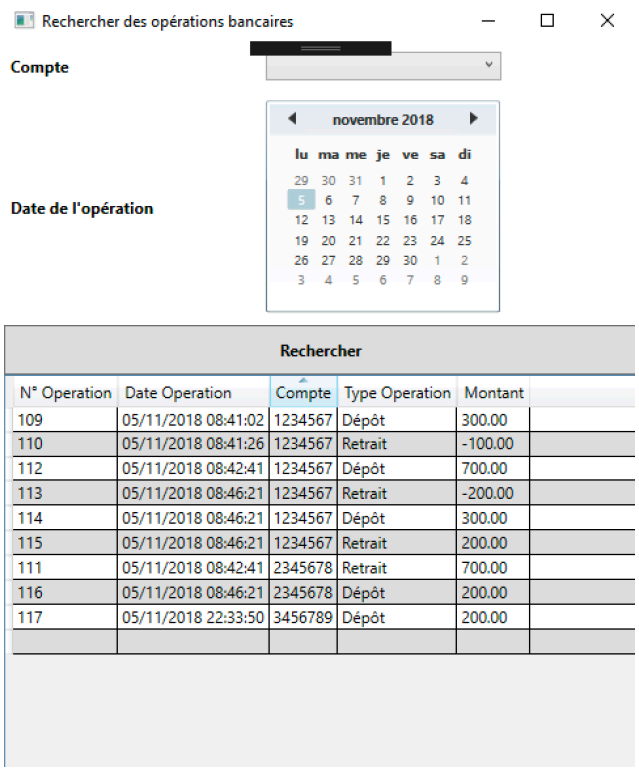


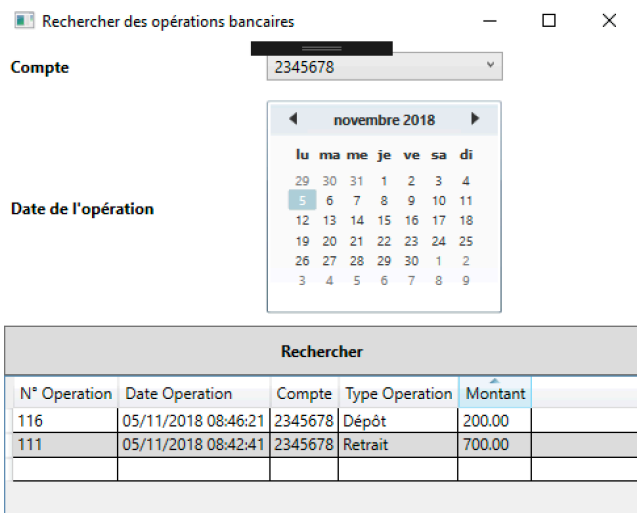
Le binding de chaque TextBox sera réalisé sur la property adéquate d'un objet `Compte`. Exemple : `<TextBox x:Name="TextBoxIdCompte" … Text="{Binding` **`CompteAjoute.IdCompte`**`}"/>`. Vous devrez instancier l'objet `Compte` dans le constructeur.

- Ajouter une fenêtre (ou page) permettant de rechercher des opérations bancaires (utiliser un *DataGrid* pour afficher les opérations) en fonction d'une date et/ou d'un compte.
  Indications :
  - Ajouter la méthode de recherche des opérations dans le DAL `IQueryable<Operation> RechercherOperations(int? idCompte, DateTime? dateOperation)` (vous pourrez utiliser la clause Linq `Where` : http://dotnetpattern.com/linq-where-operator, http://www.tutorialsteacher.com/linq/linq-filtering-operators-where). `DateTime?` permet de passer une date/heure null (https://stackoverflow.com/questions/221732/datetime-null-value). Idem pour `int?`.
  - Si vous utilisez un contrôle `Calendar` et ne sélectionnez pas de date, une date sera tout de même renvoyée (`01/01/0001 00:00:00`) par son attribut `SelectedDate` :



  - Pour ne récupérer que la date d'un `DateTime`, vous pouvez utiliser la méthode `ToShortDateString()`.
  - Pour récupérer la valeur du `DateTime?` : `dateOperation.`**`Value`**.
  - Le `DateTime` et le `DateTime?` contenant un jour/mois/année heures/minutes/secondes, le plus simple sera de comparer le jour, le mois et l'année : `o => o.DateOperation.`**`Day`** `== dateOperation.`**`Value.Day`** `&& o.DateOperation.`**`Month`** `== dateOperation.`**`Value.Month`** `&& …`
  - Exemples d'affichage :





- Dans les affichages précédents nous avons dû spécifier les colonnes du DataGrid afin de ne pas afficher la propriété de navigation qui contient un objet `Compte`. Nous avons également spécifié le format d'affichage de la date.

```
<DataGrid x:Name="DataGridOperations" Grid.Row="3"
        CanUserReorderColumns="True" CanUserResizeColumns="True"
            CanUserResizeRows="False" CanUserSortColumns="True"
                    ItemsSource="{Binding Operations}"
                         AutoGenerateColumns="False"
                        AlternatingRowBackground="Gainsboro"  AlternationCount="2">
                <DataGrid.Columns>
                    …
                    <DataGridTextColumn Header="Date Operation" IsReadOnly="True"
Binding="{Binding DateOperation, StringFormat=\{0:dd/MM/yyyy HH:mm:ss\}}"/>
                    …
                </DataGrid.Columns>
</DataGrid>
```

- Ajouter une fenêtre permettant de rechercher les virements en fonction d'un compte donné (qu'il soit au débit ou au crédit). Même principe que pour la recherche des opérations.