M3105 – Conception et programmation objet avancée

TD5 Partie 1 – Tests unitaires & TDD (Suite)

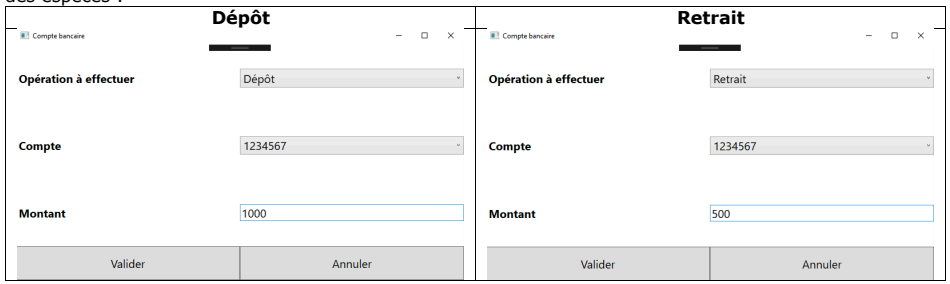
INFO

Objectifs :

Réaliser une application WPF en appliquant les bonnes pratiques suivantes :

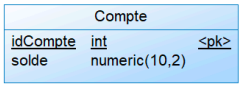
- En utilisant une architecture à 3 couches
- Basé sur l'utilisation de vues SQL
- Implémentant des tests unitaires
- Exécutant des procédures stockées

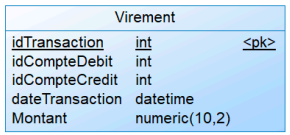
L'application permet de simuler un distributeur automatique de billets sur lequel on peut déposer ou retirer des espèces :

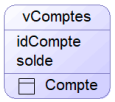


1- Création de la base de données

Exécuter le script nommé `Script ComptesBancaires SQL Server.sql` dans SQL Server Management Studio. 2 tables sont créées (`Compte` et `Virement`) ainsi qu'une vue.





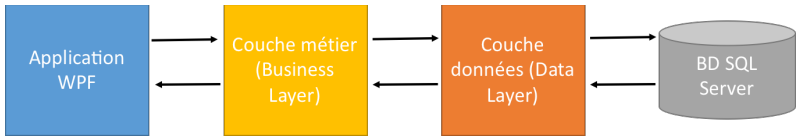


BONNE PRATIQUE : *Les vues seront utilisées en lecture (SELECT). Comme indiqué en CM, il est préférable d'exécuter des requêtes sur des vues, plutôt que directement sur des tables (moins de problèmes de verrous => critère de performance ; découplage entre l'application et la base => critère de maintenabilité).*
De même en écriture (insert, update, delete), il est préférable d'utiliser des procédures stockées pour éviter l'injection SQL (Cf. dernière section) ou des requêtes paramétrées (https://webman.developpez.com/articles/aspnet/sqlparameter/csharp/).

2- Création des projets

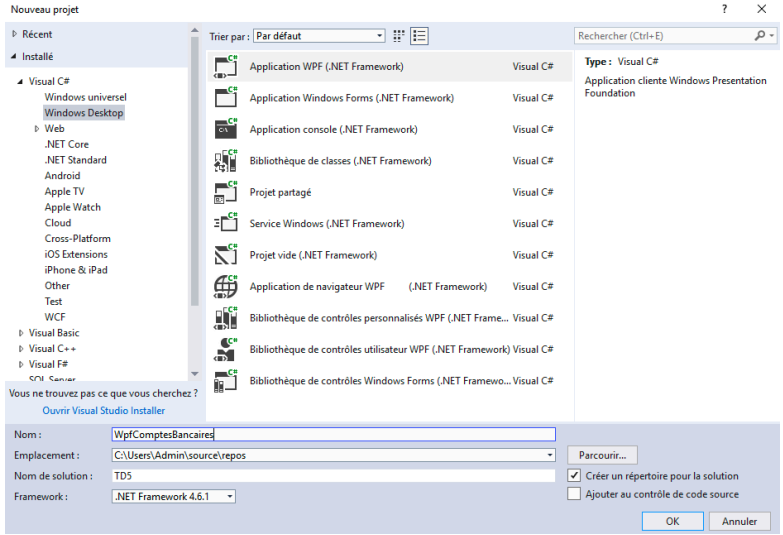
L'application à réaliser est ainsi structurée en 3-tiers :

1. `DataLayer` : classes d'accès aux données.
2. `BusinessLayer` contenant les objets métier et accédant à la couche 1.
3. `WpfComptesBancaires` représentant l'application WPF accédant à la couche 2.



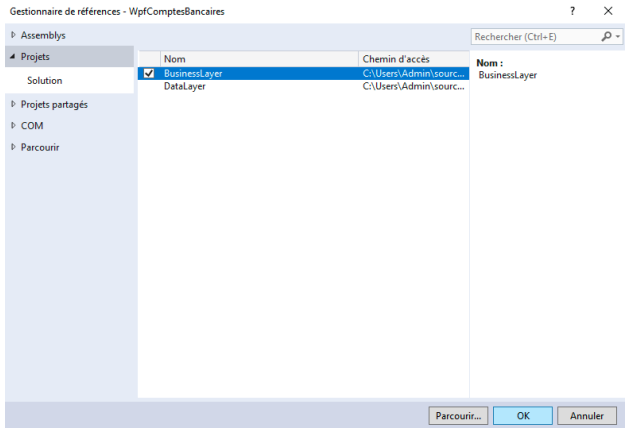
BONNE PRATIQUE : *Il est préconisé d'appliquer le patron MVVM pour développer une application utilisant une base de données ou à défaut de la structurer comme ci-dessus.*

Créer un projet « Application WPF (.NET Framework) » nommé `WpfComptesBancaires` et sa solution.

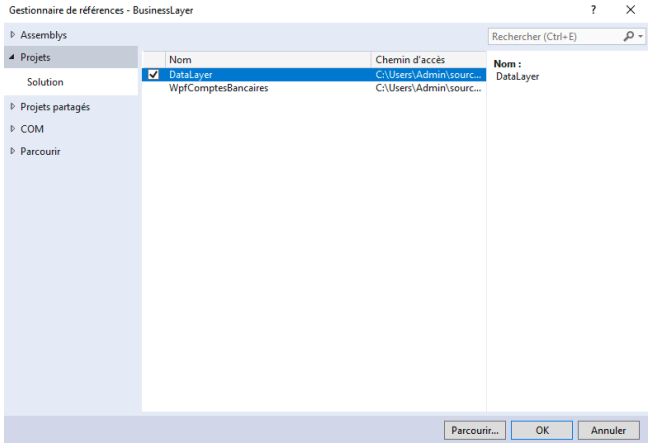


Ajouter un projet `DataLayer` de type « Bibliothèque de classes (.NET Framework) » à la solution.
Ajouter un projet `BusinessLayer` de type « Bibliothèque de classes (.NET Framework) » à la solution.

Ajouter la référence vers `BusinessLayer` dans `WpfComptesBancaires`.



Ajouter la référence vers `DataLayer` dans `BusinessLayer`.



Exécuter l'application.

3- Développement de la couche `DataLayer`

- Supprimer la classe créée.
- Cliquer avec le bouton droit de la souris sur le projet `DataLayer` puis *Ajouter > Elément existant* et ajouter le fichier `DataAccess.cs` disponible sur le serveur.
- Modifier la chaine de connexion située dans la méthode `OpenConnection() : Data Source=srv-jupiter;Database=monlogin`

ATTENTION au namespace du fichier, si vous n'avez pas respecté le nom du projet.

Remarque : la classe d'accès aux données utilise le mode déconnecté pour récupérer les données des tables (méthode `GetData`) : on se connecte à la base de données, on récupère les données dans une table de données locale (`DataTable`) puis on se déconnecte. On pourrait aussi utiliser le mode connecté (comme en M2104).

4- Développement de la couche `BusinessLayer`

Supprimer la classe créée.

Créer les classes `ServiceCompte` et `Compte`.
Dans la classe métier `Compte` (il s'agit d'une **classe "normale"** qui va représenter une entité), créer les *property* correspondant aux colonnes de la vue `vComptes`, un constructeur sans paramètre et un constructeur paramétré (2 paramètres de type `Int32` et `Double`).

La classe `ServiceCompte` accède à la couche `DataLayer`. Coder pour le moment la méthode `public List<Compte> GetAllComptes()` permettant de récupérer tous les comptes de la BD :

```
        /// <summary>
        /// Récupère la liste de tous les comptes (id et solde) de la base de données
        /// </summary>
        /// <returns>Liste des comptes bancaires</returns>
        public List<Compte> GetAllComptes()
        {
            …
        }
```

- Exécute la méthode `GetData()` sur un objet de la classe `DataAccess` et stocke le résultat dans un objet `DataTable` local. La requête qui sera passée en paramètre de la méthode `GetData()` portera sur la vue `vComptes` (pas de ; en fin de commande SQL).
- Une boucle récupère chaque ligne du `DataTable` (collection `monDataTable.rows`) et créé un objet `Compte` à partir de chaque champ de la ligne (http://msdn.microsoft.com/fr-fr/library/system.data.datatable.rows(v=vs.110).aspx). Chaque objet est ensuite ajouté à une liste d'objets `Compte`. Penser à caster les valeurs des champs en utilisant la classe statique `Convert`.
- La liste est retournée.
- Penser à utiliser un `try catch` et à lever une exception en cas d'erreur (ou à renvoyer `null`).

Ne pas oublier de définir la classe `public` de façon à pouvoir l'utiliser…

5- Développement de l'application WPF

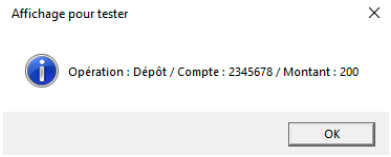
Indications :

- Pour simplifier, définir le `DataContext` sur la fenêtre (Cf. TD4 ou CM).
- Liste des opérations :

  - Créer une property de type `ObservableCollection<string>`.
  - Initialiser la collection aux valeurs *Retrait* et *Dépôt*.
  - Binder l'attribut `ItemsSource` de la ComboBox à la property.
- Liste des comptes :

  - Créer une property de type `ObservableCollection<Compte>`.
  - Binder l'attribut `ItemsSource` de la ComboBox à la property. Indiquer que vous affichez la property `IdCompte` de la classe `Compte` dans la ComboBox en utilisant l'attribut `DisplayMemberPath`.
  - Créer un objet de type `ServiceCompte` et appeler la méthode `GetAllComptes`.
  - Pour transformer une `List` en `ObservableCollection` : `MaPropertyObservableCollection = new ObservableCollection<Compte>(MaListe)`
- Montant :
  - Vous devez savoir faire… Attention, le binding se fait dans un seul sens, il n'est donc pas utile d'implémenter l'interface `INotifyPropertyChanged`.
- Bouton « Valider » :

- Pour le moment, coder uniquement un affichage pour tester :



- Penser à utiliser l'attribut `SelectedItem` des ComboBox pour récupérer les valeurs sélectionnées et à créer les properties nécessaires.

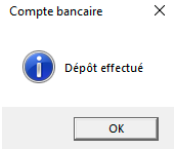
Exécuter l'application.

Pour pouvoir réaliser le débit, il faut ajouter la méthode `SetDebitCredit` dans la classe `ServiceCompte`.

```
        /// <summary>
        /// Met à jour le solde du compte passé en paramètre en fonction du montant passé en
paramètre. Si montant<0 => débit, sinon crédit.
        /// </summary>
        /// <param name="c">Compte à débiter ou créditer</param>
        /// <param name="Montant">Montant du débit ou crédit</param>
        /// <returns>Résultat de la mise à jour (update) du solde du compte : true => réussi,
false => échec</returns>
        public bool SetDebitCredit(Compte c, Double Montant)
        {
            …
        }
```

Cette méthode appelle la méthode `SetData` de la classe `DataAccess` et lui passe en paramètre un ordre *update* permettant de mettre à jour le solde (débit ou crédit) du compte sélectionné dans la base de données. Cette méthode retourne la valeur (true ou false) retournée par `SetData`.

Ajouter ensuite le code du bouton « Valider ». Afficher un message si l'opération (retrait / dépôt) a bien été effectuée. De même, dans le cas contraire, ajouter un message d'erreur.



Vérifier dans la base de données la mise à jour du solde.

Pour le codage du bouton « Annuler », voir en fin de TP.

6- Codage des tests

Codage des tests de `GetAllComptes`

La méthode `GetAllComptes` exécute une instruction SELECT sur la vue `vComptes` de la base de données et créée une liste de `Compte`. Il faut donc qu'il y ait toujours les mêmes comptes (mêmes id & solde) dans la BD de façon que les tests fonctionnent toujours, même si l'on ajoute des comptes et/ou modifie les soldes. Il paraît donc important avant d'effectuer tout test :

- de supprimer tous les comptes existants
- d'insérer des comptes qui seront attendus dans les tests de la méthode.

Ainsi, coder la méthode `InitialisationDesTests` permettant de supprimer les lignes de la table `Compte` (`delete…`) et d'ajouter 2 comptes : (1234567, 1000) et (2345678, 2000). Cette méthode utilisera à chaque fois (pour la suppression et les 2 insertions) la méthode `SetData` de la classe `DataAccess`.

```
        /// <summary>
        /// Cette méthode appelle la méthode SetData sur un objet de type DataAccess pour
supprimer les données de la table Compte
        /// et insérer 2 comptes bancaires : 1234567 / 1000 et 2345678 / 2000.
        /// </summary>
        [TestInitialize()]
        public void InitialisationDesTests()
        {
            …
        }
```

Rappel : la méthode `TestInitialize` sera exécutée lors de chaque test.

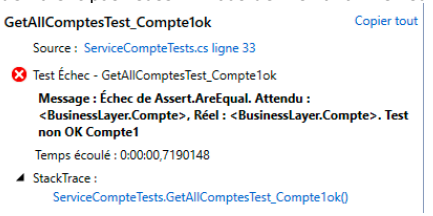
Remarque : penser à ajouter une référence à `DataLayer` dans le projet de test.

Coder ensuite 2 méthodes permettant de tester que le 1er compte de la liste retournée par `GetAllComptes` est bien le compte (1234567, 1000) attendu. De même pour le second (2345678, 2000).

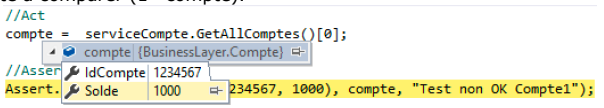
```
        [TestMethod()]
        public void GetAllComptesTest_Compte1ok()
        {
            …
        }

        [TestMethod()]
        public void GetAllComptesTest_Compte2ok()
        {
            …
        }
```

Pour comparer 2 objets, utiliser `Assert.AreEqual()` : https://msdn.microsoft.com/fr-fr/library/ms243496.aspx

Lancer les tests. Les tests ne devraient pas réussir. Vous devriez avoir l'erreur suivante.



On voit bien que la méthode `GetAllComptesTest_Compte1ok` compare 2 comptes à priori identiques (1, 1000) mais que la méthode `AreEqual()` renvoie `false`. Vous pouvez le vérifier en mettant un point d'arrêt sur la première ligne de code de la méthode `GetAllComptesTest_Compte1ok` puis menu *Test* > *Déboguer* > *Tous les tests*. Continuer ensuite le débogage en mode pas à pas et vérifier le contenu de la variable récupérant le compte à comparer (1er compte).



Cela est dû au fait que les 2 objets résidant dans des espaces mémoires différents ne sont pas égaux. Il est nécessaire de coder la méthode `Equals` (comme l'an dernier !) dans la classe `Compte`, de façon que l'on puisse comparer les 2 objets.

```
        public override bool Equals(object obj)
        {
            …
        }
```

Relancer les tests. Cette fois, ils devraient réussir.

Améliorer les tests

Plutôt que de tester chaque objet, MSTest (et MSTestv2) permet de comparer une collection d'objets (par exemple, une liste) à une collection espérée (ici, "rentrée" en dur), en utilisant la classe `CollectionAssert`. Créer une nouvelle procédure de test nommée `GetAllComptesTest_TousLesComptesOK()` utilisant `CollectionAssert.AreEqual()` (https://msdn.microsoft.com/fr-fr/library/ms243721.aspx).

Remarque : vous pouvez garder les 2 tests précédemment créés

Remarque : nous n'avons pas testé l'exception retournée par la méthode `GetAllComptes`. Pour la lever, il faut supprimer soit la vue `vComptes`, soit la table `Compte`, ou que le chaine de connexion ne soit pas valide. Dans tous les cas, les 3 tests réalisés précédemment seront en échec. Il n'est donc pas vraiment utile de tester l'exception (même si on peut le faire).

Codage des tests de `SetDebitCredit`

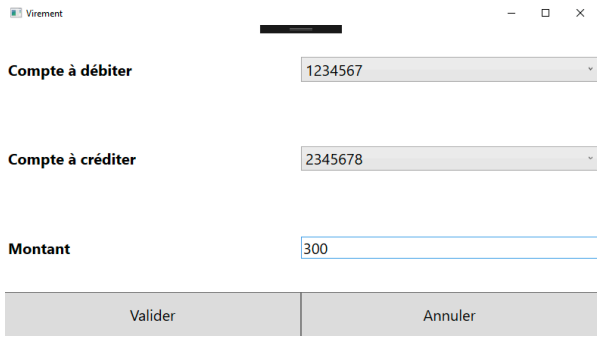
Créer un premier test nommé `SetDebitCreditTest_AvecDebitOK` testant le débit. Vous testerez la valeur retournée par le booléen.
Créer un second test nommé `SetDebitCreditTest_AvecCreditOK` testant le crédit.
Exécuter les tests.

Améliorer les tests
Ici, nos tests ne sont pas suffisants, car ils ne permettent pas de vérifier que le solde a bien été mis à jour dans la base de données. Modifier les tests pour qu'ils vérifient le nouveau solde du compte passé en paramètre.

7- Codage du bouton « Annuler » de l'application WPF

Afin de simplifier, seul le montant sera remis à zéro (procéder comme pour le TD4 pour notifier la vue).

8- Virement



Ajouter un nouveau projet WPF nommé `WpfVirement`.

Comme nous avons (bien) codé l'application précédente en couches, ce nouveau projet utilisera également les couches `BusinessLayer` et `DataLayer`.
Nous aurions également pu ajouter une nouvelle page au projet `WpfComptesBancaires`, mais il s'agit ici de démontrer que notre application a été bien architecturée…

Ajouter une référence à `BusinessLayer`.

Réaliser le code du virement en respectant les principes du TDD.

Indications :
- Squelette de la méthode de virement de `BusinessLayer` :

```
        /// <summary>
        /// Réalise un virement du compte en paramètre n°1 vers le compte en paramètre n°2 d'un montant en paramètre n°3
        /// </summary>
        /// <param name="compteDebit">Compte à débiter</param>
        /// <param name="compteCredit">Compte à créditer</param>
        /// <param name="Montant">Montant du virement</param>
        /// <returns>Résultat de la mise à jour (update) des soldes des 2 comptes : true => réussi, false => échec</returns>
        public bool Virement (Compte compteDebit, Compte compteCredit, Double Montant)
        {
        }
```

- Exemple de commande SQL à envoyer en un seul appel au SGBD (gestion des transactions) :

```
Begin transaction; DECLARE @error INT=0; Update compte set solde = solde-500 where idcompte=1234567; SET @error = @error + @@error; Update compte set solde = solde+500 where idcompte=2345678; SET @error = @error + @@error; if @error=0 commit else rollback
```

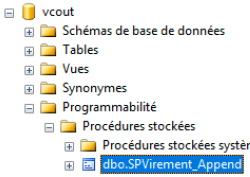
```
-- Ne fonctionne pas => ROLLBACK
Begin transaction;
DECLARE @error INT=0;
Update compte set solde = solde-100 where idcompte=1234567;
SET @error = @error + @@error;
Update compte set solde = solde+10000000000000000000 where idcompte=2345678;
SET @error = @error + @@error;
IF @error = 0 --Si error est égale à 0, donc s'il n'y a eu aucune erreur
    COMMIT
ELSE
    ROLLBACK


-- OK => Commit
Begin transaction;
DECLARE @error INT=0;
Update compte set solde = solde-100 where idcompte=1234567;
SET @error = @error + @@error;
Update compte set solde = solde+100 where idcompte=2345678;
SET @error = @error + @@error;
IF @error = 0
    COMMIT
ELSE
    ROLLBACK
```

Remarques :
- *Le code précédent est du code T-SQL, spécifique à Microsoft SQL Server.*
- *`@@error` est une variable d'environnement récupérant l'erreur levée lors de la dernière exécution. `SET` permet d'affecter une variable. Les variables déclarées en T-SQL commencent toujours par `@` (un seul).*
- *Dans ce cas, il aurait été grandement préférable de créer une procédure stockée qui exécute le code précédent en lui passant les bons paramètres (Cf. section suivante).*

9- Améliorations

- Le client n'a pas d'autorisation de découvert (débit et virement impossibles). Vous ajouterez d'abord 2 méthodes de test (1 pour le débit et 1 pour le virement) puis modifierez le code des méthodes à tester en conséquence (TDD). Vous lèverez une exception de type `ArgumentException` ou mieux votre propre exception (Cf. http://www.infoworld.com/article/3010009/application-development/implementing-a-custom-exception-class-in-c.html, https://docs.microsoft.com/en-us/dotnet/standard/exceptions/how-to-create-user-defined-exceptions).

Pour les plus rapides :
- A chaque fois qu'un virement est effectué, il est enregistré dans la base de données (table `Virement`).
  - Exécuter le code de la procédure stockée du fichier `procedure stockee Virement SQLServer.sql` dans SQL Server Management Studio.
  - On peut voir la procédure stockée dans SQL Server :



  - Mettre en commentaire le code à l'intérieur de la méthode `Virement`.
  - Code de l'appel de la procédure (à intégrer dans le fichier `DataAccess.cs`) :

```
        /// <summary>
```

```
        /// Permet d'exécuter la procédure stockée de virement (mises à jour
des soldes des comptes de débit et de crédit et enregistrement du virement)
        /// </summary>
        /// <param name="idCompteDebit">Id du compte débité</param>
        /// <param name="idCompteCredit">Id du compte crédité</param>
        /// <param name="montant">Montant du virement</param>
        /// <returns>Retourne un booléen si le PS a réussi ou non.</returns>
        public bool VirementBancaire(int idCompteDebit, int idCompteCredit,
Double montant)
        {
            try
            {
                bool retour=false;
                if (this.OpenConnection())
                {
                    SqlCommand cmd = new SqlCommand("SPVirement_Append",
this.connection);
                    cmd.CommandType = CommandType.StoredProcedure;
                    //Création des paramètres
                    SqlParameter pIdCompteDebit =
cmd.Parameters.Add("@IDCOMPTEDEBIT", SqlDbType.Int);
                    pIdCompteDebit.Direction = ParameterDirection.Input;
                    SqlParameter pIdCompteCredit =
cmd.Parameters.Add("@IDCOMPTECREDIT", SqlDbType.Int);
                    pIdCompteCredit.Direction = ParameterDirection.Input;
                    SqlParameter pMontant = cmd.Parameters.Add("@MONTANT",
SqlDbType.Float);
                    pMontant.Direction = ParameterDirection.Input;
                    SqlParameter pRetVal = cmd.Parameters.Add("@RETOUR",
SqlDbType.Int);
                    pRetVal.Direction = ParameterDirection.Output;
                    //Initilisation de la valeur des paramètres aux arguments
de la méthode
                    pIdCompteDebit.Value = idCompteDebit;
                    pIdCompteCredit.Value = idCompteCredit;
                    pMontant.Value = montant;
                    //Exécution de la procédure
                    cmd.ExecuteNonQuery();
                    //Récupération de la valeur de retour
                    if ((int)pRetVal.Value == 1)
                        retour = true;
                    this.CloseConnection();
                    return retour;
                }
                else
                    return false;
            }
            catch
            {
                return false;
            }
        }
```

On est obligé de créer les paramètres `(SqlParameter)` qui seront passés à la procédure stockée. Il n'est donc pas possible de réaliser un code générique dans le cas de l'appel de procédures stockées.

- Créer le code de la méthode `Virement` appelant la méthode `ViremementBancaire`.
- ***BONNE PRATIQUE :*** *L'utilisation de procédures ou fonctions stockées empêche l'injection SQL. En outre, elles peuvent intégrer une gestion d'erreur.*

- Ajouter la méthode suivante, s'exécutant après les tests, permettant de supprimer toutes les données et insérer les données initiales contenues dans le script SQL

```
[TestCleanup()]
public void NettoyageDesTests()
{
}
```

- Compléter le code du bouton « Annuler ».