Introduction au développement en couches

Lorsque l'on est débutant en programmation, on entend souvent dire qu'il est important de développer ses applications en utilisant des couches, en séparant le code logique de l'interface utilisateur, etc.… Mais pour un débutant, toutes ces phrases sont abstraites et ne signifient rien. Au travers de cet article, nous allons voir, avec des exemples simples et concrets, comment réaliser une application utilisant ces différentes couches. Nous apprendrons leur importance, ce qu'elles représentent et comment les utiliser pour développer une application.

Table des matières

Introduction 3

Les couches 4

- L'interface graphique 4
- Les objets métier (Business Objects) 5
- La couche d'accès aux données (DAL, Data Access Layer) 6
- La couche métier (BLL, Business Logic Layer) 9

Utilisation 10

Gestion des exceptions 11

Pourquoi développer en utilisant les couches ? 13

Organisation de la solution 14

Conclusions 15

Introduction

Développer une application, tout le monde (ou presque) est capable de le faire : il suffit de concevoir l'interface utilisateur, puis de rajouter le code qui va bien sur le clic d'un bouton par exemple.

Si l'on est débutant, le premier réflexe que l'on va avoir sera d'insérer le code d'accès aux données (listes des clients, des produits, etc.…) directement sur le clic du bouton. Cette technique, bien qu'entièrement fonctionnelle, n'est pas idéale. En effet, imaginez que l'on vous impose que la liste des clients ne proviennent pas d'une base de données mais d'un fichier XML. Vous seriez alors obligé de modifier tout le code de votre application avant de la redéployer sur des postes clients. Si à l'inverse vous aviez utilisé le développement en couche, vous n'auriez qu'à modifier la couche d'accès à la liste de clients.

Une application « en couche » est généralement composée d'au moins 4 couches :

- **L'interface graphique** (**GUI**, *Graphic User Interface*)
- **Les objets métier** (*BusinnessObjects*)
- **La couche d'accès aux données** (**DAL**, *Data Access Layer*)
- **La couche métier** (**BLL**, *Business Logic Layer*)

Personnellement, j'ai également l'habitude de rajouter une couche supplémentaire (**Tools**) qui me sert à regrouper diverses classes communes aux couches (par exemple les exceptions personnalisées).

Nous allons donc détailler chacune de ces couches et je vais tâcher de vous montrer comment les implémenter. Attention, il s'agit ici des méthodes de travail que j'utilise : rien ne garantit qu'elles soient optimales car le but est de vous démontrer comment essayer de bien développer votre application.

Afin de faciliter la compréhension de chacun, nous allons partir du principe que nous développons une simple application nous permettant de lister l'ensemble des clients d'un magasin (ces informations proviennent de la base de données).

Les couches

L'interface graphique

Pour notre interface graphique, nous allons faire quelque chose de simple : une fenêtre avec une ListBox (pour afficher la liste des clients), une TextBox et 2 boutons.

En ce qui concerne la technologie utilisée, j'ai choisi **WPF** mais tout ce qui sera dit au cours de cet article s'applique également à du WindowsForms classique et à de l'ASP.NET.

Voici le code utilisé :

```
<Window x:Class="DeveloppementNTiers.GUI.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DeveloppementNTiers.GUI" Height="300" Width="300"
    Loaded="WindowLoaded"
    >
    <StackPanel Orientation="Vertical">
      <Button x:Name="btGetClients" Content="Get Clients"
Click="btGetClientsClick" />

      <ListBox x:Name="lbClients" Margin="5" ItemsSource="{Binding}" />
      <TextBox x:Name="tbClientName" Margin="5" Text="{Binding
ElementName=lbClients, Path=SelectedItem.ClientLastName}" /

      <Button x:Name="btUpdateClientName" Content="Update Client Name"
Click="btUpdateClientNameClick" Margin="5" />
    </StackPanel>
</Window>
```

Il vous faut aussi rajouter, à votre application, un fichier de configuration dans lequel vous allez indiquer la chaîne de connexion à la base de données :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="DBConnectionString"
         connectionString="Data Source=T-THOLE05\SQLEXPRESS;Initial
Catalog=AdventureWorks;Integrated Security=True"
         providerName="System.Data.SqlClient"
         />
  </connectionStrings>
</configuration>
```

Les objets métier (Business Objects)

Les objets métier correspondent à tous les objets spécifiques que vous allez manipuler. Typiquement, dans notre cas, nous allons manipuler des clients. Un client, c'est un identifiant, un nom, un prénom, etc.… Toutes ces caractéristiques pourraient très bien représenter les propriétés de notre objet **Client** :

```
public class Client
    {
        private int m_ClientID;

        public int ClientID
        {
            get { return m_ClientID; }
            set { m_ClientID = value; }
        }

        private string m_ClientLastName;

        public string ClientLastName
        {
            get { return m_ClientLastName; }
            set { m_ClientLastName = value; }
        }

        private string m_ClientFirstName;

        public string ClientFirstName
        {
            get { return m_ClientFirstName; }
            set { m_ClientFirstName = value; }
        }

        public Client(int id, string lastname, string firstname)
        {
            this.ClientID = id;
            this.ClientLastName = lastname;
            this.ClientFirstName = firstname;
        }
    }
```

Si vous aviez eu besoin de manipuler des produits, vous auriez très certainement créé une classe Product, etc.….

Dans notre cas, nous avons-nous-même écrit le code de cette classe. Généralement, on part du principe qu'une table dans la base de données représente un objet. Et que chaque colonne de la table représente une propriété de l'objet. Si vous aviez eu 50 tables dans votre base de données, vous auriez pu avoir besoin de 40 objets (certaines tables ne représentent pas forcément un objet, comme par exemple les tables temporaires, les tables système, etc.…). Pour générer le code correspondant (et non pas l'écrire à la main), vous pouvez utiliser ce que l'on appelle le « **Mapping Objet-Relationnel** », mais il s'agit là d'un autre sujet.

La couche d'accès aux données (DAL, Data Access Layer)

C'est dans cette partie que vous allez gérer tout ce qui concerne l'accès aux données. Il y a, dans cette couche, plusieurs parties bien distinctes :

- Une partie pour tout ce qui concerne l'accès à la base de données
- Une partie pour tout ce qui concerne les requêtes sur la base de données

Pour la partie « accès à la base de données », nous allons créer une classe qui se chargera de la connexion :

```
public class Sql
    {
        private DbConnection m_SqlCnx = null;

        private static Sql s_Instance;
        private static object s_InstanceLocker = new object();

        private ConnectionStringSettings m_CnxStringSettings;
        public ConnectionStringSettings CnxStringSettings
        {
            get { return m_CnxStringSettings; }
            set { m_CnxStringSettings = value; }
        }

        // Singleton
        public static Sql Instance
        {
            get
            {
                lock (s_InstanceLocker)
                {
                    if (s_Instance == null)
                    {
                        s_Instance = new Sql();
                    }

                    return s_Instance;
                }
            }
        }
```

```
        // Cette méthode créé une connexion à la BDD et la renvoit
        public DbConnection GetSqlConnection()
        {
            DbProviderFactory factory =
DbProviderFactories.GetFactory(CnxStringSettings.ProviderName);

            if (m_SqlCnx == null)
            {
                m_SqlCnx = factory.CreateConnection();
            }

            m_SqlCnx.ConnectionString = CnxStringSettings.ConnectionString;

            if (m_SqlCnx.State == System.Data.ConnectionState.Closed)
            {
                m_SqlCnx.Open();
            }

            return m_SqlCnx;
        }

        public void CloseConnection()
        {
            if (this.m_SqlCnx != null)
            {
                if (this.m_SqlCnx.State != System.Data.ConnectionState.Closed)
                {
                    this.m_SqlCnx.Close();
                }
            }
        }
    }
```

Cette classe utilise un « **design pattern** » que l'on appelle « **Singleton** » : il s'agit d'une technique utilisée pour limiter l'instanciation d'une classe à un seul objet.

Il nous faut à présent écrire la classe qui va exécuter les requêtes sur la base de données :

```
public class ClientDAO
    {
        private static ClientDAO s_Instance;
        private static object s_InstanceLocker = new object();

        // Singleton
        public static ClientDAO Instance
        {
            get
            {
                lock (s_InstanceLocker)
                {
                    if (s_Instance == null)
                    {
                        s_Instance = new ClientDAO();
                    }
```

```
                    return s_Instance;
                }
            }
        }

        public List<Client> GetClients()
        {
            List<Client> clients = null;

            using (DbConnection cnx = Sql.Instance.GetSqlConnection())
            {
                clients = GetClientsFromDB(cnx);
            }

            return clients;
        }

        private List<Client> GetClientsFromDB(DbConnection cnx)
        {
            List<Client> clients = null;

            using (DbCommand cmd = cnx.CreateCommand())
            {
                cmd.CommandType = System.Data.CommandType.Text;
                cmd.CommandText = "SELECT TOP 10 * FROM Person.Contact";

                using (DbDataReader reader = cmd.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        Client client = new Client();
                        client.ClientID = reader["ContactID"] == DBNull.Value ?
default(int) : int.Parse(reader["ContactID"].ToString());
                        client.ClientLastName = reader["LastName"] ==
DBNull.Value ? default(string) : reader["LastName"].ToString();
                        client.ClientFirstName = reader["FirstName"] ==
DBNull.Value ? default(string) : reader["FirstName"].ToString();

                        clients.Add(client);
                    }
                }
            }

            return clients;
        }
    }
```

Cette classe, que l'on a appelée **ClientDAO** (pour *Data Access Object*) est, elle aussi, composée d'un Singleton. Vous pouvez voir, dans la méthode **GetClient**, qu'elle récupère l'instance de la classe de connexion à la base de données, puis qu'elle utilise cette connexion pour effectuer une requête. Ici, j'ai utilisé une requête de type texte mais pour bien faire, il aurait fallu utiliser une procédure stockée.

La couche métier (BLL, Business Logic Layer)

Nous avons donc, d'un coté notre interface graphique et de l'autre, notre couche d'accès au données. Il serait tout à fait possible de relier directement les 2. Cependant, comment feriez-vous si vous aviez besoin d'appliquer des règles ou d'effectuer des opérations sur les résultats issus de la base de données ? Où mettriez-vous votre code ? Pas dans l'interface graphique, car il ne doit y avoir que ce qui concerne l'interface. Dans la couche d'accès aux données ? Non, car cette couche ne traite que de l'accès aux données. Où alors ?

C'est dans une nouvelle couche, que l'on appelle **BLL** (ou *Business Logic Layer*), que l'on va mettre ce code. C'est le lien entre votre interface utilisateur et votre DAL.

```
public class ClientManager
    {
        private static ClientManager s_Instance;
        private static object s_InstanceLocker = new object();

        // Singleton
        public static ClientManager Instance
        {
            get
            {
                lock (s_InstanceLocker)
                {
                    if (s_Instance == null)
                    {
                        s_Instance = new ClientManager();
                    }

                    return s_Instance;
                }
            }
        }

        public ConnectionStringSettings ConnectionString
        {
            set
            {
                Sql.Instance.CnxStringSettings = value;
            }
        }

        public List<Client> GetClients()
        {
            // Ici, on peut appliquer des règles métier
            return ClientDAO.Instance.GetClients();
        }
    }
```

Comme vous pouvez le voir, cette classe est, elle aussi, composée d'un Singleton. Et elle possède une méthode (**GetClients**) qui se contente uniquement d'appeler la méthode de la DAL !

Utilisation

A présent, nous avons bien tout ce qu'il nous faut :

- Notre GUI référence notre BLL, nos objets métier, et notre futur couche d'outils
- Notre BLL référence notre DAL, nos objets métier, et notre futur couche d'outils
- Notre DAL ne référence que nos objets métier

Il ne nous reste plus qu'à utiliser tout cela ensemble.

Pour cela, il vous suffit, dans le code votre interface graphique, de faire appel aux méthodes de votre BLL :

```
public partial class Window1 : System.Windows.Window
    {

        public Window1()
        {
            InitializeComponent();
        }

        private void WindowLoaded(object sender, RoutedEventArgs e)
        {
            // Au chargememnt de l'application
            // on indique la chaine de connexion
            BLL.ClientManager.Instance.ConnectionString =
ConfigurationManager.ConnectionStrings["DBConnectionString"];
        }

        private void btGetClientsClick(object sender, RoutedEventArgs e)
        {
            this.lbClients.DataContext = BLL.ClientManager.Instance.GetClients();
            this.lbClients.DisplayMemberPath = "ClientLastName";
        }

        private void btUpdateClientNameClick(object sender, RoutedEventArgs e)
        {
            //
        }
    }
```

Gestion des exceptions

Tout ce que l'on a fait à présent est entièrement fonctionnel : vous venez tout juste de développer votre première application utilisant la séparation des couches.

Cependant, une bonne application se doit d'avoir une gestion des exceptions.

C'est à ce niveau là qu'intervient ma fameuse couche « Tools », qui va me permettre, entre autre, de définir mes propres exceptions :

```
public class CustomException : Exception
    {
        public CustomException()
            : base()
        {
            //
        }

        public CustomException(string msg)
            : base(msg)
        {
            //
        }

        public CustomException(string msg, Exception innerEx)
            : base(msg, innerEx)
        {
            //
        }
    }
```

Il ne nous reste plus qu'à utiliser cette classe dans notre couche métier :

```
public List<Client> GetClients()
        {
            try
            {
                // Ici, on peut appliquer des règles métier
                return ClientDAO.Instance.GetClients();
            }
            catch (Exception e)
            {
                throw new CustomException("An error has occured", e);
            }
        }
```

Et lors de l'appel aux méthodes de votre BLL, attrapez les exceptions que vous lancez :

```
private void btGetClientsClick(object sender, RoutedEventArgs e)
        {
            try
            {
                this.lbClients.DataContext =
BLL.ClientManager.Instance.GetClients();
                this.lbClients.DisplayMemberPath = "ClientLastName";
            }
            catch (CustomException ce)
            {
                MessageBox.Show(ce.Message);
            }
        }
```

Pourquoi développer en utilisant les couches ?

Nous avons vu comment, d'un point de vue technique, nous pouvons mettre en place une application utilisant les différentes couches. Mais une question reste toujours en suspens : « *Pourquoi développer de cette façon* ? ».

En effet, jusqu'à maintenant, vous avez sans doute développé des applications, entièrement fonctionnelles, sans utiliser cette technique. Alors pourquoi ? On pourrait très bien penser que c'est :

- Pour faire joli ? C'est vrai qu'avec une technique comme celle-ci, l'explorateur de solutions de Visual Studio donne tout de suite l'impression d'un travail sérieux, mais je ne suis pas sur que votre employeur préfère le superflu aux résultats
- Pour faire comme les autres ? Oui, mais quel intérêt si vous ne comprenez pas ce que vous faîtes ?

En fait, il y a plusieurs avantages à utiliser cette technique :

- La maintenance des données est indépendante du support physique de stockage
- La maintenance des traitements est simplifiée
- La gestion des traitements depuis la couche de présentation est facilitée
- Le travail en équipe est optimisé
- La migration d'un environnement graphique à un autre est relativement simple

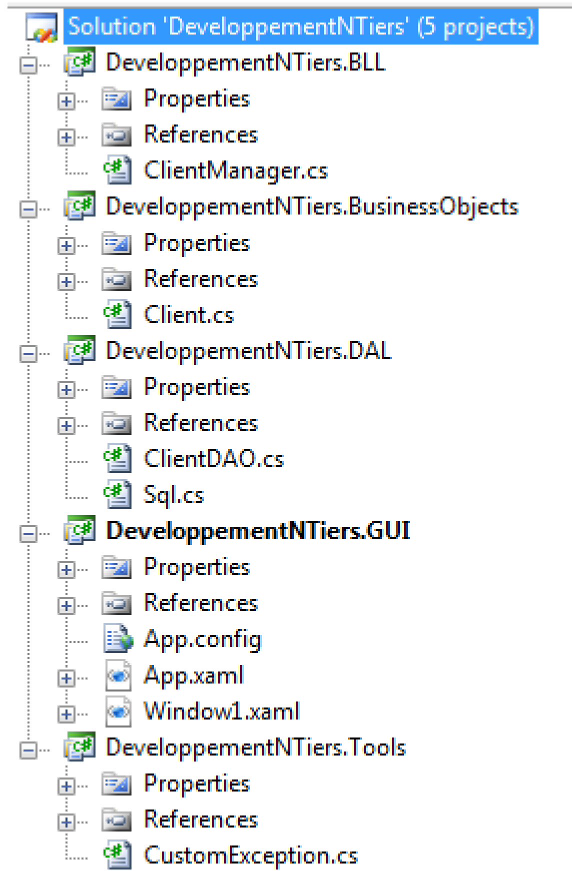
Lorsque l'on dit que le travail en équipe est optimisé, la raison est simple : alors qu'un des membres de l'équipe travaille sur la couche d'accès aux données, un autre peut tout à fait travailler sur la couche métier ou sur l'interface graphique sans perturber le travail de ses collègues.

De même, dans le cas de migration (d'interface utilisateur par exemple), là encore, la tâche est simplifiée. Ainsi, inutile de redévelopper tout ce qui a été fait jusqu'à maintenant : il vous suffit de modifier l'interface.

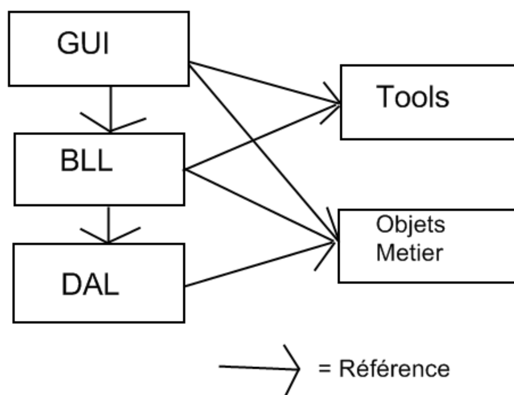
Dans le cas de notre article, nous sommes partit d'une application WPF pour tout ce qui était interface graphique. Si demain, je souhaitais utiliser la même chose, mais dans une application Web, vais-je devoir tout reprendre ? Bien sur que non : grâce à ce découpage en couche, je serais en mesure de garder la couche d'accès aux données et la couche métier, et de ne changer que la couche GUI (passer de WPF à ASP.NET). Une fois sur ma nouvelle interface graphique, je n'aurais qu'à reprendre les appels à ma BLL (couche métier) et le tour est joué ☺

Organisation de la solution

Voici un aperçu de la solution Visual Studio représentant votre projet :



En ce qui concerne le référencement, je vous le rappelle via ce petit dessin :



Conclusions

Comme vous avez pu le voir, développer une application utilisant les différentes couches (DAL, BLL, etc.…) se révèle extrêmement simple à partir du moment où l’on sait exactement ce que l’on doit faire (ce que j’ai tenté de vous démontrer dans cet article).

Bien sûr, d’autres concepts pourraient être abordés et certains pourraient être vus plus en détails, peut-être dans un prochain article ☺