

1. Le pattern Factory

Patron mobilisé pour la création des objets.

Le but de ce pattern est de permettre à une application cliente de pouvoir interagir avec des objets ayant les mêmes responsabilités mais facilement substituables entre eux.

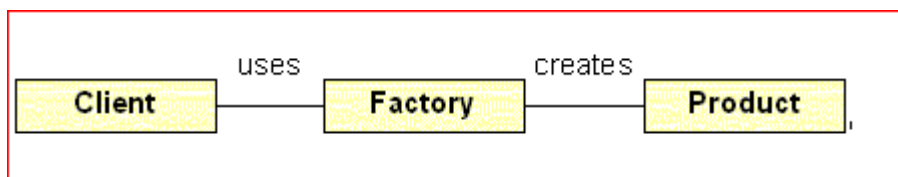
Le Pattern Factory est l'un des modèles de création le plus utilisé.

Le nom de ce modèle est directement inspiré à la réalité de sa fonction, car il spécialise un objet unique à la fabrique (création) d'autres objets.

Comme avec les autres modèles ou pattern, il y a un certain nombre de variations du modèle de fabrique de classe, bien que la plupart des variantes utilisent le même jeu d'acteurs primaires, un Client, une Fabrique et un Produit (Client, Factory, Product).

Le client est un objet qui requiert une instance d'un autre objet (le produit). Plutôt que de créer une instance du produit directement, le client délègue cette responsabilité à la fabrique de classe. Une fois invoquée, la fabrique crée une nouvelle instance du produit et la retourne au client.

Plus simplement, le client utilise la fabrique pour créer une instance du produit.



Un exemple pour illustrer ce pattern et le fait que le client n'a pas à connaître le produit concret.

Soient deux produits concrets et une classe abstraite dont ils héritent.

```
public abstract class ProduitAbstrait
{
    public abstract void Methode();
}
public class ProduitConcret1 : ProduitAbstrait
{
    public override void Methode()
    {
        Console.WriteLine("Produit 1");
    }
}
public class ProduitConcret2 : ProduitAbstrait
{
    public override void Methode()
    {
        Console.WriteLine("Produit 2");
    }
}
```

Voici maintenant la classe dite Factory qui permettra de créer les produits concrets via la méthode GetProduit().

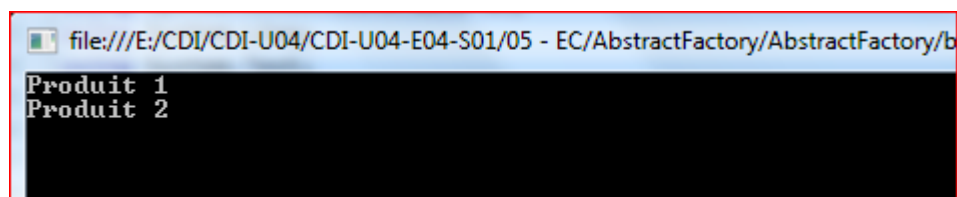
Cette méthode de fabrique reçoit sous forme d'une chaîne le nom du type de produit à créer et retourne un produit complet.

```
public static class Fabrique
{
    public static ProduitAbstrait getProduit(string typeProduit)
    {
        switch (typeProduit)
        {
            case "Pro1":
                return new ProduitConcret1();
            case "Pro2":
                return new ProduitConcret2();
            default:
                return null;
        }
    }
}
```

Le test :

```
class Program
{
    static void Main(string[] args)
    {
        ProduitAbstrait a = Fabrique.getProduit("Pro1");
        a.Methode();
        ProduitAbstrait b = Fabrique.getProduit("Pro2");
        b.Methode();
        Console.ReadKey();
    }
}
```

Le résultat :



```
file:///E:/CDI/CDI-U04/CDI-U04-E04-S01/05 - EC/AbstractFactory/AbstractFactory/b
Produit 1
Produit 2
```

Nous voyons ainsi que le client peut utiliser un produit sans le connaître : seul le type abstrait est connu.

Cette approche permet de pouvoir faire évoluer les produits concrets sans retoucher le client. Ceci est valable tant que les interfaces de la classe abstraite ne changent pas.

2. Un exemple au niveau de la couche DAL

Nous allons ici programmer la couche d'accès aux données en nous appuyant sur une fabrique mise à notre disposition.

La classe DbProviderFactories permet de créer des fournisseurs de données qui s'appuient sur nombre de fournisseur (extrait) :

```
System.Data.EntityClient .EntityProviderFactory
System.Data.Odbc .OdbcFactory
System.Data.OleDb .OleDbFactory
System.Data.OracleClient .OracleClientFactory
System.Data.SqlClient .SqlClientFactory
```

Le client n'a pas besoin de connaître quel type de fournisseur est créé.

Toutes ces classes concrètes héritent d'une classe abstraite DbProviderFactory :

```
❖ CreateCommand()
❖ CreateCommandBuilder()
❖ CreateConnection()
❖ CreateConnectionStringBuilder()
❖ CreateDataAdapter()
❖ CreateDataSourceEnumerator()
❖ CreateParameter()
❖ CreatePermission(System.Security.Permissions.PermissionState)
❖* DbProviderFactory()
🔧 CanCreateDataSourceEnumerator
```

Ce mécanisme permet de ne pas devoir modifier le client si la base cible ou le fournisseur change.

Il nous faudra préciser, lors de la création de la connexion, le nom du fournisseur.

Notre classe DB doit donc exposer une propriété supplémentaire ProviderName dont la valeur sera fournie par l'application.

```
public class DB
{
1 référence
public static string ConnectionString
{ get; set; } = null;
1 référence
public static string ProviderName { get; set; } = null;
```

Et la méthode de création de Connexion évolue avec le recours au mécanisme de fabrication auquel est communiqué un élément textuel qui permet de choisir le fournisseur (ProviderName) extrait du fichier de configuration de l'application.

```
public DbConnection GetDBConnection()
{
    DbProviderFactory fabrique =
        DbProviderFactories.GetFactory(ProviderName);
    DbConnection oCon = fabrique.CreateConnection();
    oCon.ConnectionString = ConnectionString;
    oCon.Open();
    return oCon;
}
```

Il sera nécessaire de revoir les méthodes de la couche DAL afin d'utiliser les types qui se trouvent dans l'espace Data.Common en lieu et place de ceux du client SQL.

- DbConnection / SqlConnection
- DbCommand / SqlCommand
- DbParameter / SqlParameter
- DbDataReader / SqlDataReader

Exemple de méthode d'extraction d'un adhérent :

```
public Adherent GetByID(string idAdherent)
{
    using (DbConnection cnx = DB.Instance.GetDBConnection() )
    using (DbCommand command = cnx.CreateCommand())
    {
        command.CommandType = CommandType.Text;
        DbParameter pIdAdherent = command.CreateParameter();
        pIdAdherent.ParameterName = "@IdAdherent";
        pIdAdherent.DbType = DbType.String;
        pIdAdherent.Direction = ParameterDirection.InputOutput;
        pIdAdherent.Value = idAdherent;
        command.Parameters.Add(pIdAdherent);
        command.CommandText = "Select IdAdherent,PrenomAdherent,NomAdherent " +
            "from Adherent where idAdherent = @IdAdherent";
        using (DbDataReader rd = command.ExecuteReader())
        {
            return rd.Read() ? ChargerDonnees(rd) : null;
        }
    }
}
```

Il est nécessaire de produire un code SQL qui soit compatible avec l'ensemble des SGBD cibles et non propre à un dialecte particulier.

Les fonctionnalités nécessitant le recours à des techniques ou des fonctions internes du SGBD seront développées au sein de procédures stockées ou autres composants hébergés sur le serveur.

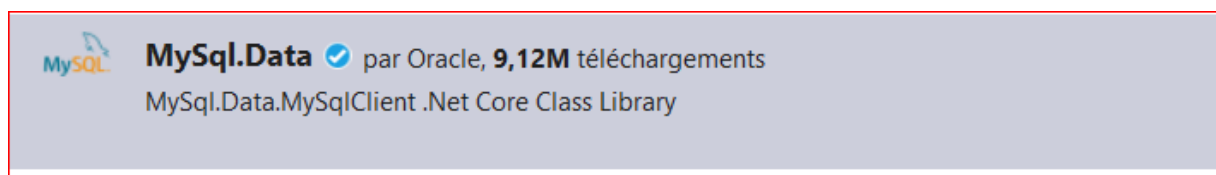
Le fournisseur du SGBD doit bien sûr avoir été préalablement installé sur le poste de travail et lié à l'application.

Celui de SQL Server est présent par défaut.

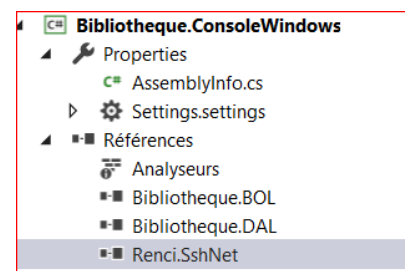
Pour installer le package d'installation du fournisseur pour MySQL, utilisez le gestionnaire de packages Nuget.

Vous pouvez prendre l'ensemble des composants avec MySQL.Data ou choisissez uniquement les composants pour la fabrique.

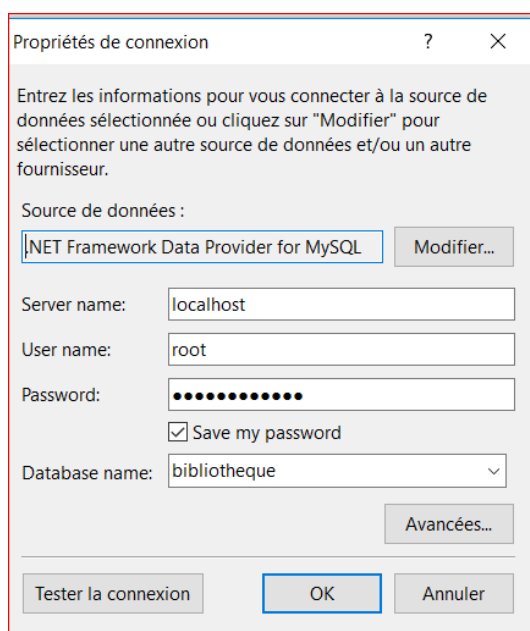
Installez ces composants au niveau de la couche DAL.



A noter : Il est nécessaire de disposer de la DLL Renci.SshNet au niveau de l'application. Sinon, vous obtenez une erreur lors de la connexion à la base MySQL. La solution de facilité réside dans l'installation complète du package nuget MySqlData au niveau de l'application... Pas très logique mais cette solution de facilité fonctionne....



Ajout d'une chaîne de connexion



Extraction des informations relatives à la chaîne de connexion :

Ajouter au niveau du projet d'application, une référence à la DLL System.Configuration puis recourir à la classe utilitaire ConfigurationManager :

```
ConnectionStringSettings parametresConnexion =  
    ConfigurationManager.ConnectionStrings["BibliothequeCSSQL"];  
DB.ConnectionString = parametresConnexion.ConnectionString;  
DB.ProviderName = parametresConnexion.ProviderName;
```