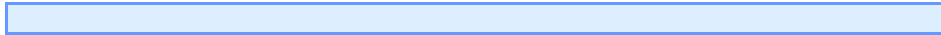


Principes de réalisation d'un DAC indépendant du SGBD en .NET 2.0

par

Date de publication : 14/12/2005

Dernière mise à jour : 14/12/2005



- I - Présentation
- II - Principes (en .NET 1.1)
 - A - Définition simplifiée d'une interface
 - B - Définition de la classe fabrique d'objets
 - C - Utilisation
- III - L'utilisation en .NET 2.0
 - A - Hiérarchie des classes et des interfaces
 - B - Nouvelles classes intégrées au Framework 2.0
 - C - Où sont stockées les informations sur les providers?
 - D - Enumération des providers installés
 - E - Utilisation
 - F - La classe DbConnectionStringBuilder
- IV - Conclusion

I - Présentation

De plus en plus, les applications doivent être utilisables avec toute une série d'éléments extérieurs. Comme principaux éléments, on peut bien entendu citer les bases de données. Le changement de support au niveau de la base de données ne peut en aucun cas nécessiter une réécriture complète du code. C'est pour cette raison qu'il est préférable de passer par une couche intermédiaire pour accéder aux données. Cette fonctionnalité, nous allons la décrire ici, que ce soit pour le framework 1.1 ou pour le 2.0.

Cet article est axé principalement sur le framework 2.0. Cependant, pour améliorer la compréhension du fonctionnement, des explications sont données sur les techniques utilisées en 1.1. Pour plus de détails sur la manière de créer un accès aux données indépendant du provider, je vous renvoie vers <http://nx.developpez.com/articles/dac/>

Vous pouvez télécharger ce tutoriel au format PDF à l'adresse suivante: <http://ditch.developpez.com/dotnet/factories/pdf/factories.pdf>

II - Principes (en .NET 1.1)

A - Définition simplifiée d'une interface

Une interface est en quelque sorte un contrat que doivent respecter les classes qui l'implémentent. L'objectif est de signaler qu'une classe doit implémenter des méthodes dont le nom est décrit dans le contrat. Ainsi, cette classe doit pouvoir effectuer certaines opérations, quelque soit la manière de les réaliser. De plus, lors de la compilation, si une classe ne propose pas les fonctionnalités du "contrat" qu'elle est censée respecter, le compilateur renverra une erreur.

Prenons un exemple de la vie de tous les jours : Certaines personnes sont des développeurs, d'autres des pilotes de formule1, #

Quelque soit notre occupation et notre formation, nous sommes tous des personnes. C'est pourquoi nous allons imaginer deux classes (Développeur et PiloteF1) qui doivent implémenter l'interface `Personne`. Par ailleurs, le Pilote de F1 est également un pilote, nous allons donc créer l'interface correspondant à sa fonction.

Les interfaces:

```
public interface Personne
{
    string Nom ;
    string Prénom ;
    #

    public char[2] getInitiales() ;
}

public interface Pilote
{
    public string getEcurie() ;
}
```

Les classes qui implémentent ces interfaces:

```
public class Développeur : Personne
{
    string Nom ;
    string Prénom ;
    HashTable LangagesConnus ;
    #

    public char[2] getInitiales() ;
}

public class PiloteF1 : Personne, Pilote
{
    string Nom ;
    string Prénom ;
    string Ecurie;
    #

    public char[2] getInitiales() ;
    public string getEcurie() ;
}
```

On pourrait ainsi comparer une interface avec une classe abstraite pour laquelle aucune implémentation n'a été réalisée.

B - Définition de la classe fabrique d'objets

Pour réaliser notre couche d'accès aux données, il est nécessaire de passer par les interfaces. Effectivement, grâce à celles-ci nous connaissons les méthodes implémentées et nous sommes sûrs qu'elles sont bel et bien implémentées. C'est pourquoi il suffit (note de l'auteur: "si, si il suffit...") de créer une classe implémentant des méthodes statiques. Ces méthodes retournent des objets qui implémentent les interfaces. Pour savoir quel type d'objets doivent être renvoyés, nous passerons un paramètre aux différentes méthodes utilisées. Il existe bien entendu toute une série de moyens pour éviter de passer le paramètre à chaque méthode.

Certains se disent alors " Oui mais alors il faut passer de nouveau le code en revue pour y mettre le nom de l'objet désiré ". Evidemment, le problème serait le même, c'est pourquoi il est bon de mettre le nom dans le Web.Config dans le cadre d'asp.NET ou le App.config pour les WinForms.

Premier exemple: SqlConnection, OleDbConnection, OracleConnection et bien d'autres implémentent tous l'interface IDbConnection.

```
public class DbTemplate
{
    public static IDbConnection Connection(string sType)
    {
        IDbConnection cn=null;
        switch(sType)
        {
            case "SqlServer": cn=new System.Data.SqlClient.SqlConnection();
                break;
            case "Access": cn=new System.Data.OleDb.OleDbConnection();
                break;
            //...
        }
        return cn;
    }

    public static IDbCommand Command(string sType)
    {
        IDbCommand cmd=null;
        switch(sType)
        {
            case "SqlServer": cmd=new System.Data.SqlClient.SqlCommand();
                break;
            case "Access": cmd=new System.Data.OleDb.OleDbCommand();
                break;
            //...
        }
        return cmd;
    }
}
#
```

On pourrait bien entendu ajouter d'autres méthodes renvoyant d'autres types d'objet. Il ne reste plus qu'à l'utiliser de la manière suivante :

```
IDbConnection Conn=DbTemplate.Connection("SqlServer");
IDbCommand Cmd=DbTemplate.Command("SqlServer");
IDataReader dr = DbTemplate.DataReader("SqlServer");

Cmd.Connection=Conn;
#
dr = Cmd.ExecuteReader() ;
#
```

C - Utilisation

Avec l'utilisation d'une clé dans le Web.Config (asp.NET) :

```
using System.Configuration ;
#

string DbType = ConfigurationSettings.AppSettings["DbType"] ;*

IDbConnection Conn=DbTemplate.Connection(DbType);
IDbCommand Cmd = DbTemplate.Command(DbType);
IDataReader dr = DbTemplate.DataReader(DbType);

Cmd.Connection=Conn;
#
dr = Cmd.ExecuteReader() ;
#
```

On notera que cette technique nécessite d'ajouter manuellement les différents providers dans la classe explicitée précédemment.

Certes, cette technique permet de mettre à jour uniquement la couche d'accès aux données, ce qui est plus efficace que de devoir mettre à jour tous les accès aux données.

III - L'utilisation en .NET 2.0

A - Hiérarchie des classes et des interfaces

Comme dans le cas de l'utilisation en .NET 1.1, les différents providers doivent implémenter des interfaces. C'est grâce à ces interfaces qu'il est possible de réaliser une telle organisation.

B - Nouvelles classes intégrées au Framework 2.0

Le framework 2.0 inclut toute une série de nouvelles classes. Parmi celles-ci, deux nous intéressent tout particulièrement:

La première est `DbProviderFactory`. Il s'agit de la classe permettant de créer des objets `Command`, `Connection` ainsi que tous ceux qui s'y rapportent. Nous verrons cela dans le paragraphe consacré à l'utilisation de la classe `DbProviderFactory`.

Cette première classe est obtenue à l'aide d'une seconde, `DbProviderFactories`. Concrètement, c'est elle qui renvoie les objets des différents providers selon le paramètre qui lui est passé. C'est pourquoi les méthodes de cette classe sont statiques (comme l'étaient les méthodes de notre classe fabrique en 1.1).

C - Où sont stockées les informations sur les providers?

La liste des providers disponibles est accessible dans le `machine.config` situé dans le répertoire `$winapp$\Microsoft.NET\Framework\v2.0.50727\CONFIG`.

En voici un extrait:

```
<system.data>
  <DbProviderFactories>
    <add name="Odbc Data Provider" invariant="System.Data.Odbc" description=".Net _
      Framework Data Provider for Odbc" type="System.Data.Odbc.OdbcFactory, System.Data,
    -
      Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
    <add name="OleDb Data Provider" invariant="System.Data.OleDb" description=".Net _
      Framework Data Provider for OleDb" type="System.Data.OleDb.OleDbFactory,
System.Data, _
      Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
    <add name="OracleClient Data Provider" invariant="System.Data.OracleClient" _
      description=".Net Framework Data Provider for Oracle" _
      type="System.Data.OracleClient.OracleClientFactory, System.Data.OracleClient, _
      Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
    <add name="SqlClient Data Provider" invariant="System.Data.SqlClient" description=_
      ".Net Framework Data Provider for SqlServer"
type="System.Data.SqlClient.SqlClientFactory, _
      System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
    <add name="SQL Server CE Data Provider" invariant="Microsoft.SqlServerCe.Client" description=_
      ".NET Framework Data Provider for Microsoft SQL Server 2005 Mobile Edition" _
      type="Microsoft.SqlServerCe.Client.SqlCeClientFactory, Microsoft.SqlServerCe.Client,
    -
      Version=9.0.242.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" />
  </DbProviderFactories>
</system.data>
```

D - Enumération des providers installés

Bien entendu, il n'est pas nécessaire d'utiliser les API pour accéder aux valeurs du machine.config. La classe DbProviderFactories contient une méthode permettant de récupérer la liste des providers installés sur la machine et ce dans un DataTable. Vous pouvez ainsi afficher ces données dans un GridView par exemple ou encore effectuer des opérations dessus.

```
protected sub Page_Load()
{
    DataTable dt = DbProviderFactories.GetFactoryClasses();
    gridView.DataSource = dt;
    gridView.DataBind();
}
```

Ainsi, on peut manipuler le DataTable afin de vérifier l'existence d'un provider. Cette méthode le permet:

```
if (DbProviderFactories.GetFactoryClasses.Select("InvariantName='" + invariantName & "'").Length = 0)
    Response.Write(invariantName + "n'existe pas");
```

E - Utilisation

Comme on peut le voir dans l'exemple suivant, l'utilisation des classes DbProviderFactory et DbProviderFactories ressemble fortement à l'utilisation de notre objet DbTemplate en .NET 1.1.

```
IDataReader dbreader = null;

DbProviderFactory dbfactory =
    DbProviderFactories.GetFactory(
        ConfigurationSettings.AppSettings.Get("Provider"));

IDbConnection dbconn = dbfactory.CreateConnection();
dbconn.ConnectionString = ConfigurationSettings.AppSettings.Get("ConnectionString");
try
{
    dbconn.Open();
    IDbCommand dbcomm = dbconn.CreateCommand();
    dbcomm.Connection = dbconn;
    dbcomm.CommandText = "SELECT * FROM Clients";
    dbreader = dbcomm.ExecuteReader(CommandBehavior.CloseConnection);
    while (dbreader.Read())
    {
        Console.WriteLine("Client ID:{0}",
            dbreader["ClientId"].ToString());
    }
}
finally
{
    if (dbreader != null) dbreader.Dispose();
}
```

Pour l'exemple précédent, il est nécessaire d'inclure deux namespaces:

```
using System.Data.Common;
using System.Configuration;
```

F - La classe DbConnectionStringBuilder

Même si cet objet n'est pas indispensable, il en est pour le moins intéressant. Il permet de diminuer le risque d'erreur lorsque vous écrivez votre ConnectionString.

Son utilisation est simple comme on peut le voir sur l'exemple suivant:

```
String connectionString = String.Empty;

//Get serverName from the user

SqlConnectionStringBuilder conStrbuilder = new SqlConnectionStringBuilder();
conStrbuilder.DataSource = serverName;
conStrbuilder.UserID = uid;
conStrbuilder.Password = pwd;

SqlConnection c = new SqlConnection (conStrbuilder.ConnectionString);
```

Toutes les propriétés ont une valeur par défaut. MinPoolSize est à 0 par exemple.

L'utilisation de cet objet permet de vérifier si il n'y a pas d'erreurs lors de la compilation en lieu et place de les avoir à l'exécution.

IV - Conclusion

Nous venons de voir comment rendre une application indépendante d'un provider et le fonctionnement des objets utilisés. J'espère que cet article vous permettra d'élargir l'utilité de vos applications.

Remarque: l'utilisation d'une classe fabrique n'est pas "la" solution dans tous les cas. Effectivement, les différents systèmes de bases de données n'implémentent pas tous de la même manière le langage SQL. Il s'agit donc tout de même de réécrire les requêtes pour certaines de ces bases de données.

Il faudrait, pour être certain d'avoir des applications nécessitant le moins de modifications possibles lors du portage de la connexion d'une base de données vers une autre, réaliser une classe fabrique pour les requêtes SQL.

On notera également que l'utilisation des procédures stockées permet dans bien des cas de solutionner les problèmes. Malheureusement, tous les SGBD n'implémentent pas le support des procédures stockées.

Merci à David Pédehourcq pour l'aide apportée pour la réalisation de cet article.