


# LINQ to SQL

Nous avons vu que LINQ sait parfaitement manipuler les objets et leurs propriétés, et est en cela un outil idéal pour la manipulation de données. Pourtant, ces collections de données ont un inconvénient majeur : la fin de l'exécution de l'application implique leur perte complète. La solution la plus courante pour conserver ces données entre plusieurs lancements de l'application consiste à utiliser une base de données à laquelle est confiée la tâche de sauvegarde de ces informations.

Malgré leurs similitudes syntaxiques, les requêtes LINQ et SQL ne sont pas compatibles. LINQ to SQL contient une implémentation de fournisseur de données pour la base de données SQL Server. Ce fournisseur permet la traduction des requêtes LINQ en langage SQL. Un autre problème est que LINQ n'est capable de manipuler que des collections d'objets. Or les bases de données ne manipulent jamais d'objets. Pour résoudre ce problème, la solution est de créer des classes représentant les données de la base de données. Cette technique est nommée **mappage objet-relationnel**.

 La base de données utilisée par tous les exemples de cette section est la base Northwind de Microsoft. Le script SQL permettant sa création est disponible en téléchargement depuis la page Informations générales.

## 1. Le mappage objet-relationnel

LINQ to SQL propose trois solutions pour créer les classes C# représentant les données contenues dans la base de données.

La première de ces solutions est la création manuelle. Cette solution est la plus accessible, mais également la plus fastidieuse. L'écriture manuelle de code C# pour refléter la base de données est ainsi très souvent réservée aux modifications minimales. Il est également possible de générer ce code à l'aide de l'outil en ligne de commande **SQLMetal**, ou même d'utiliser le **concepteur visuel objet-relationnel** intégré à Visual Studio.

### a. Utilisation de SQLMetal

L'outil SQLMetal est une application en ligne de commande livrée avec Visual Studio. Trois types de génération peuvent être effectués avec cet outil :

- La génération des classes et des attributs de mappage à partir d'une base de données existante.
- La génération d'un fichier intermédiaire de mappage (.dbml).
- La génération du code source et des attributs de mappage associés à partir d'un fichier intermédiaire de mappage (.dbml).

SQLMetal propose un grand nombre d'options pour exécuter ces différentes actions. Les principales sont détaillées ci-dessous.

#### Options de connexion

**/server:<nom>**

Nom ou adresse IP du serveur de base de données.

**/database:<nom>**

Spécifie le nom de la base de données.

**/user:<nom>**

Nom d'utilisateur SQL Server avec lequel la connexion doit être faite. Si aucune valeur n'est donnée, c'est le mode d'authentification Windows qui est utilisé.

**/password:<mot de passe>**

Mot de passe associé au nom d'utilisateur SQL Server.

**/conn:<chaîne de connexion>**

Chaîne de connexion complète à utiliser pour la connexion à la base de données. Cette option peut être utilisée à la place des quatre options précédentes.

**/timeout:<durée en secondes>**

Indique la durée maximum d'attente de la connexion au serveur. Une valeur à zéro indique une durée illimitée.

### **Options de sortie**

**/dbml:<nom du fichier>**

Génère un fichier intermédiaire de mappage au format DBML.

**/code:<nom du fichier>**

Génère le code source dans le fichier indiqué.

### **Options de génération**

**/language:<identifiant de langage>**

Indique le langage à utiliser pour la génération du code source. Les valeurs acceptées sont `vb` et `csharp`.

**/namespace:<espace de noms>**

Définit l'espace de noms dans lequel doit être placé le code généré.

**/context:<nom de type>**

Définit le nom de la classe de contexte générée. Par défaut, ce nom est déduit du nom de la base de données.

**/entitybase:<nom de type>**

Définit une classe de base pour toutes les classes d'entités. Par défaut, elles n'ont aucune classe de base.

La dernière option disponible correspond au nom du fichier de mappage à utiliser pour la génération des classes. Cette option n'est pas utilisée lorsque la génération est effectuée à partir de la base de données.



Ces options peuvent être retrouvées en exécutant la commande `SQLMetal /?`.

Les exemples suivants montrent la mise en œuvre de chacun des trois modes de génération.

### **Génération de classes à partir de la base de données**

La ligne de commande suivante génère le code source C# associé à la base de données locale Northwind en utilisant l'authentification Windows.

```
Sqlmetal /server:localhost\SQLEXPRESS /database:Northwind  
/language:csharp /code:Northwind.cs
```

Le fichier Northwind.cs généré compte environ 3700 lignes. Le diagramme de classes associé au code de ce fichier est le suivant.

**Northwind**  
Classe  
→ DataContext

⊕ Champs

⊖ Propriétés

- Categories
- CustomerCustomerDemo
- CustomerDemographics
- Customers
- Employees
- EmployeeTerritories
- OrderDetails
- Orders
- Products
- Region
- Shippers
- Suppliers
- Territories

⊕ Méthodes

**Customers**  
Classe

⊕ Champs

⊖ Propriétés

- Address
- City
- CompanyName
- ContactName
- ContactTitle
- Country
- CustomerCustomerDemo
- CustomerID
- Fax
- Orders
- Phone
- PostalCode
- Region

⊕ Méthodes

⊖ Événements

- PropertyChanged
- PropertyChanging

**Categories**  
Classe

**CustomerCustomerDemo**  
Classe

**CustomerDemographics**  
Classe

**Employees**  
Classe

**EmployeeTerritories**  
Classe

**OrderDetails**  
Classe

**Orders**  
Classe

**Products**  
Classe

**Region**  
Classe

**Shippers**  
Classe



La base de données compte treize tables et le fichier généré compte quatorze classes. Cette différence est due au fait que le code généré inclue une classe de contexte qui fournit l'accès à chacune des autres classes. Ce type, nommé Northwind par défaut, hérite de la classe `System.Data.Linq.DataContext`.

### **Génération d'un fichier .dbml à partir de la base de données**

La génération d'un fichier .dbml crée un fichier au format XML qui décrit la relation entre la base de données et les classes associées qui seront générées.

La commande suivante génère ce fichier pour la base de données Northwind :

```
Sqlmetal /server:localhost\SQLEXPRESS /database:Northwind  
/dbml:Northwind.dbml
```

L'extrait ci-dessous présente la description de la table Customers dans ce fichier.

```
<Table Name="dbo.Customers" Member="Customers">  
  <Type Name="Customers">  
    <Column Name="CustomerID" Type="System.String"  
DbType="NChar(5) NOT NULL" IsPrimaryKey="true" CanBeNull="false" />  
    <Column Name="CompanyName" Type="System.String"  
DbType="NVarChar(40) NOT NULL" CanBeNull="false" />  
    <Column Name="ContactName" Type="System.String"  
DbType="NVarChar(30)" CanBeNull="true" />  
    <Column Name="ContactTitle" Type="System.String"  
DbType="NVarChar(30)" CanBeNull="true" />  
    <Column Name="Address" Type="System.String"  
DbType="NVarChar(60)" CanBeNull="true" />  
    <Column Name="City" Type="System.String"  
DbType="NVarChar(15)" CanBeNull="true" />  
    <Column Name="Region" Type="System.String"  
DbType="NVarChar(15)" CanBeNull="true" />  
    <Column Name="PostalCode" Type="System.String"  
DbType="NVarChar(10)" CanBeNull="true" />  
    <Column Name="Country" Type="System.String"  
DbType="NVarChar(15)" CanBeNull="true" />  
    <Column Name="Phone" Type="System.String"  
DbType="NVarChar(24)" CanBeNull="true" />  
    <Column Name="Fax" Type="System.String"  
DbType="NVarChar(24)" CanBeNull="true" />
```

```

    <Association Name="FK_CustomerCustomerDemo_Customers"
Member="CustomerCustomerDemo" ThisKey="CustomerID"
OtherKey="CustomerID" Type="CustomerCustomerDemo" DeleteRule="NO
ACTION" />
    <Association Name="FK_Orders_Customers" Member="Orders"
ThisKey="CustomerID" OtherKey="CustomerID" Type="Orders"
DeleteRule="NO ACTION" />
  </Type>
</Table>

```

Par défaut, le nom d'une propriété C# est le même que celui de la colonne à laquelle il est associé. Le fichier de mappage peut être modifié pour que le code C# corresponde à des normes de codage particulières, ou pour que les noms de propriétés soient francisés, par exemple. Le code XML résultant de la mise en œuvre de ce dernier cas est le suivant :

```

<Table Name="dbo.Customers" Member="Clients">
  <Type Name="Clients">
    <Column Name="CustomerID" Member="IdClient"
Storage="_CustomerID" Type="System.String" DbType="NChar(5) NOT
NULL" IsPrimaryKey="true" CanBeNull="false" />
    <Column Name="CompanyName" Member="Societe"
Storage="_CompanyName" Type="System.String" DbType="NVarChar(40)
NOT NULL" CanBeNull="false" />
    <Column Name="ContactName" Member="NomContact"
Storage="_ContactName" Type="System.String" DbType="NVarChar(30)"
CanBeNull="true" />
    <Column Name="ContactTitle" Member="TitreContact"
Storage="_ContactTitle" Type="System.String"
DbType="NVarChar(30)" CanBeNull="true" />
    <Column Name="Address" Member="Adresse"
Storage="_Address" Type="System.String" DbType="NVarChar(60)"
CanBeNull="true" />
    <Column Name="City" Member="Ville" Storage="_City"
Type="System.String" DbType="NVarChar(15)" CanBeNull="true" />
    <Column Name="Region" Type="System.String"
DbType="NVarChar(15)" CanBeNull="true" />
    <Column Name="PostalCode" Member="CodePostal"
Storage="_PostalCode" Type="System.String" DbType="NVarChar(10)"
CanBeNull="true" />
    <Column Name="Country" Member="Pays" Storage="_Country"
Type="System.String" DbType="NVarChar(15)" CanBeNull="true" />
    <Column Name="Phone" Member="Telephone" Storage="_Phone"
Type="System.String" DbType="NVarChar(24)" CanBeNull="true" />
    <Column Name="Fax" Type="System.String"
DbType="NVarChar(24)" CanBeNull="true" />
    <Association Name="Customers_CustomerCustomerDemo"
Member="CustomerCustomerDemo" ThisKey="IdClient"
OtherKey="CustomerID" Type="CustomerCustomerDemo" />
    <Association Name="Customers_Orders" Member="Orders"
ThisKey="IdClient" OtherKey="CustomerID" Type="Orders" />
  </Type>
</Table>

```

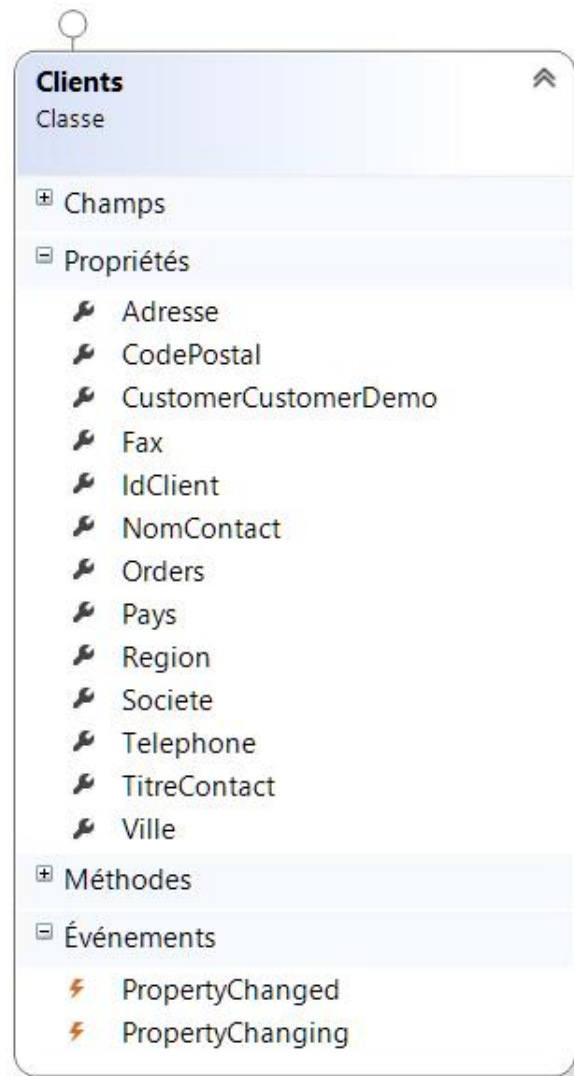
Pour chaque colonne dont le nom devrait être francisé, un attribut Member a été ajouté de manière à modifier le nom de la propriété associée. L'attribut Storage ajouté sur ces colonnes précise quant à lui le nom de la variable interne contenant une valeur associée à la colonne. Par défaut, le nom de cette variable et de la propriété correspondante ont un nom associé au nom de la colonne. Lorsque le nom de la propriété est modifié, il est nécessaire d'ajouter un mappage explicite pour le nom de la variable interne.

### **Génération de classes à partir d'un fichier .dbml**

La génération des classes C# à partir du fichier .dbml modifié dans le fichier Northwind\_fr.cs est effectuée en exécutant la commande suivante :

```
Sqlmetal Northwind.dbml /language:csharp  
/code:Northwind_fr.cs
```

Le code généré contient maintenant une classe Clients correspondant à la table Customers.



Cet outil a l'inconvénient de ne pouvoir générer des classes que pour une base de données complète. De plus, la modification manuelle de ces classes ou du fichier intermédiaire de mappage est fastidieuse et peut être source d'erreurs. Pour ces raisons, Visual Studio intègre un concepteur visuel pour le mappage objet/relationnel qu'il est

très souvent préférable d'utiliser.

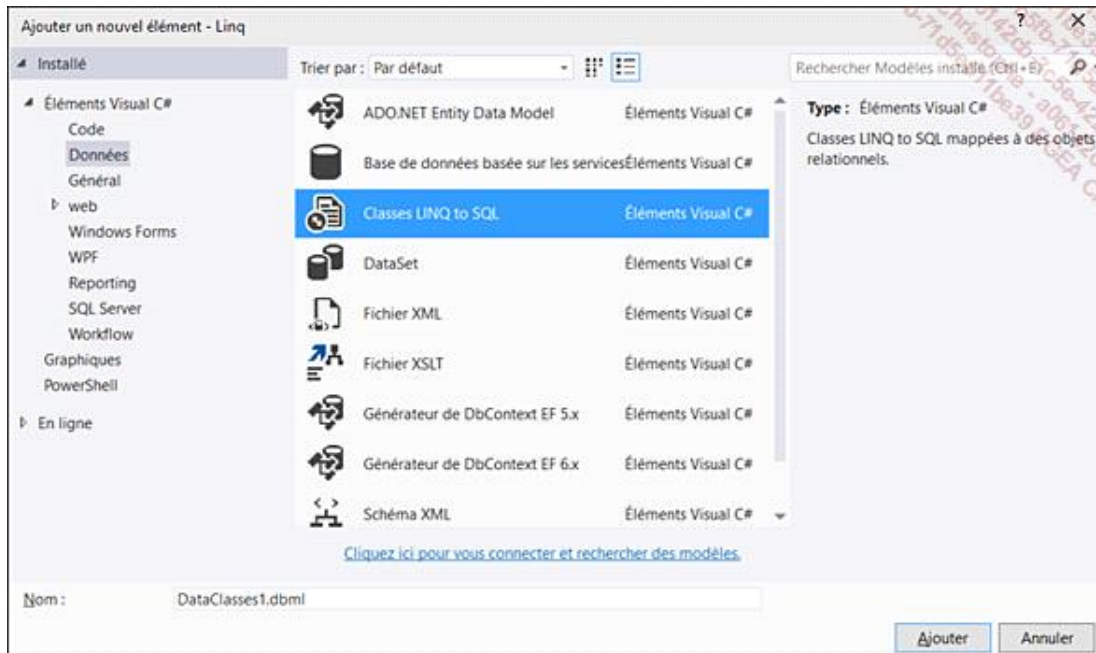
## b. Utilisation du concepteur objet/relationnel

Le concepteur visuel objet/relationnel offre une approche plus simple à prendre en main pour la génération du modèle objet associé à une base de données. Il offre la possibilité de générer un modèle partiel, ainsi que la création de méthodes permettant l'utilisation de fonctions et procédures stockées dans la base de données.

Il a néanmoins l'inconvénient de ne fonctionner qu'avec le moteur SQL Server à partir de sa version 2000. Il n'autorise également le mappage qu'entre une table (ou une vue) et une classe C#. Il n'est pas possible de créer une classe liée au résultat d'une jointure entre tables.

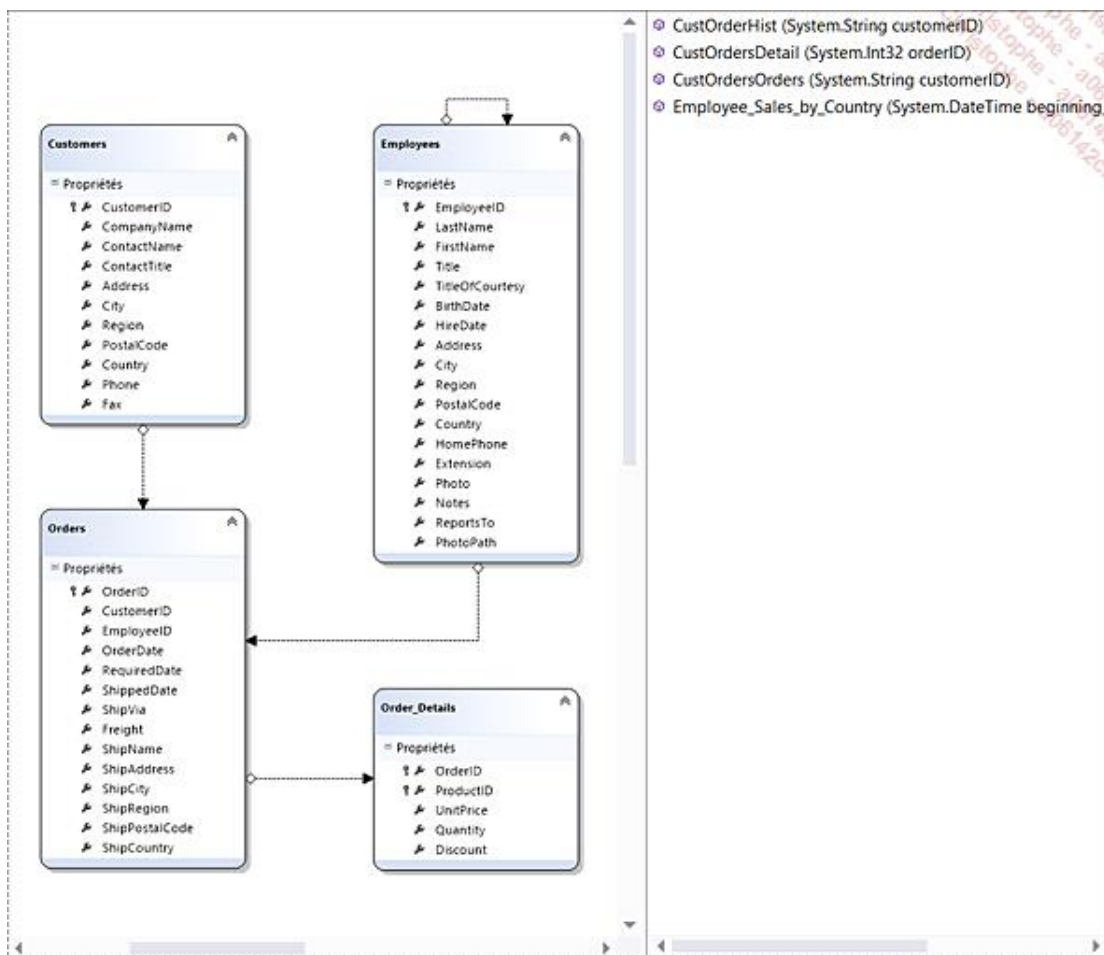
Le concepteur fonctionne également dans un mode unidirectionnel. Les modifications effectuées dans le code C# généré ne sont en effet pas répercutées dans le concepteur. Elles seront d'ailleurs complètement perdues lors la génération suivante des classes de mappage. La solution apportée par Microsoft à ce dernier problème est l'utilisation de classes et de méthodes partielles.

L'outil de conception visuelle fournit une représentation graphique des données de mappage enregistrée dans un fichier au format DBML. Ce type de fichier peut être ajouté à un projet en sélectionnant le modèle **Classes LINQ to SQL** dans la boîte de dialogue d'ajout de fichier.



L'outil de conception visuelle est ouvert automatiquement lors de l'ajout d'un fichier de type **Classes LINQ to SQL** à un projet. Sa surface est composée de deux parties. La zone située à gauche contient la représentation des classes incluses dans le mappage, tandis que la zone de droite affiche la liste des méthodes associées à des fonctions et procédures stockées.



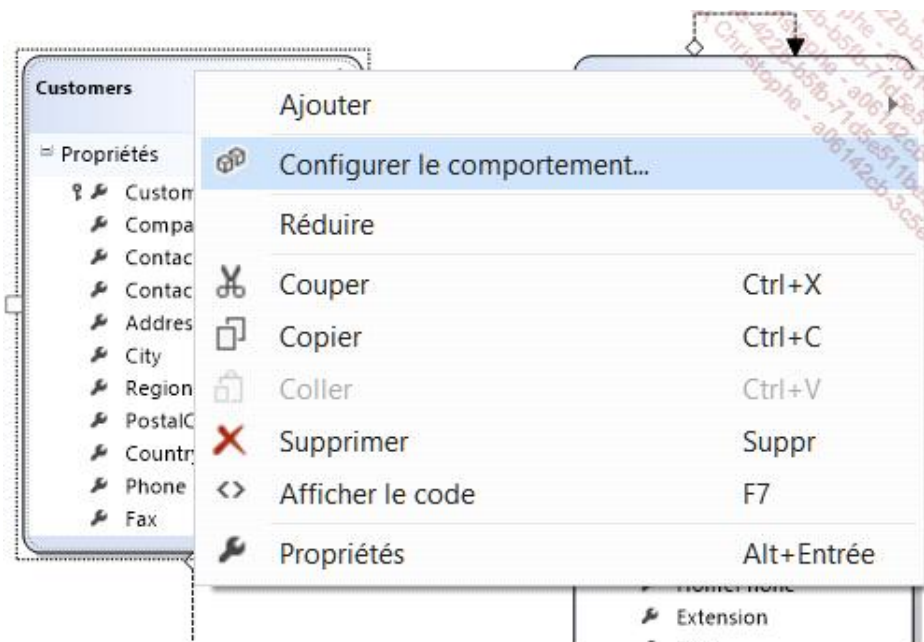


## Ajout de classe

L'ajout d'une classe représentant une table de la base de données est effectué par un glisser-déposer d'une ou plusieurs tables à partir de l'**explorateur de serveurs** vers la partie gauche de l'outil de conception visuelle. L'ajout du premier élément configure une connexion associée à l'objet `DataContext`. L'ajout d'une classe provoque la génération du code de la classe. Il déclenche également la génération du code permettant l'association de la classe et de la table dans le `DataContext`.

Le code généré inclut l'initialisation d'un objet à partir d'un enregistrement de la base de données, ainsi que la répercussion des modifications qui lui sont apportées dans la base de données. Par défaut, LINQ to SQL se base sur la structure de la table ainsi que sur sa clé primaire pour effectuer la lecture et la mise à jour des données. Les actions de lecture, de modification et de suppression peuvent néanmoins être configurées pour utiliser des procédures stockées mappées dans le `DataContext`. Ceci permet d'isoler une partie de la logique qui concerne uniquement la base de données.

Le menu contextuel de chaque classe en cours d'édition dans le concepteur visuel permet d'ouvrir la fenêtre **Configurer le comportement**.



Cette fenêtre de configuration offre la possibilité de choisir le comportement par défaut ou un comportement personnalisé pour chaque couple classe/action.

**Configurer le comportement**

Sélectionnez une classe et un comportement. Puis, choisissez soit de laisser le système générer le code automatiquement au moment de l'exécution, soit de personnaliser à l'aide des méthodes de mise à jour, d'insertion ou de suppression spécifiques.

Classe : Customers

Comportement: Supprimer

☐ Utiliser le runtime  
Laissez le système générer automatiquement la logique d'insertion, de mise à jour et de suppression au moment de l'exécution.

☒ Personnaliser

SupprimerClient (System.Int32 idClient)

Arguments de méthode	Propriétés de classe
idClient	CustomerID (Current)

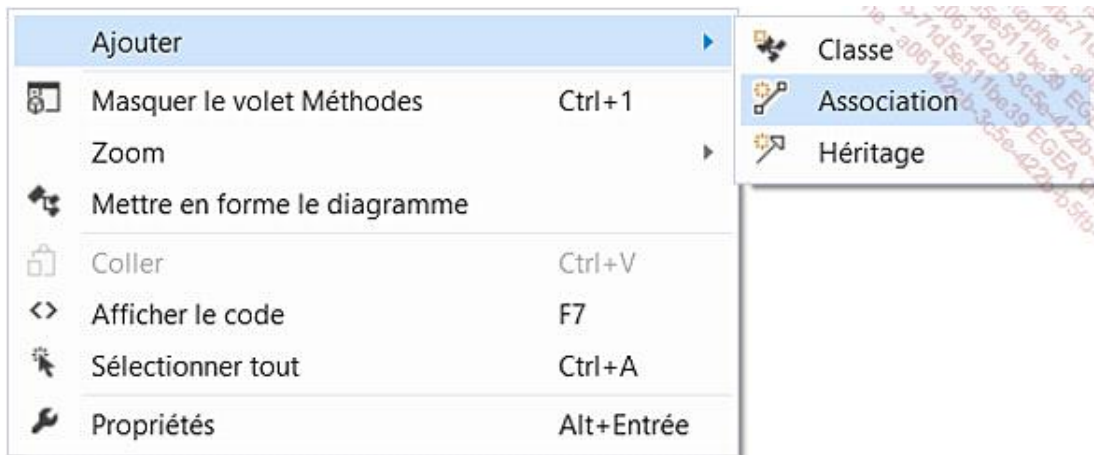
OK Annuler Appliquer

Après avoir sélectionné une procédure stockée, il est nécessaire d'indiquer comment doivent être renseignés ses paramètres. Les valeurs disponibles correspondent aux propriétés de la classe, dans leur version originale ou actuelle.

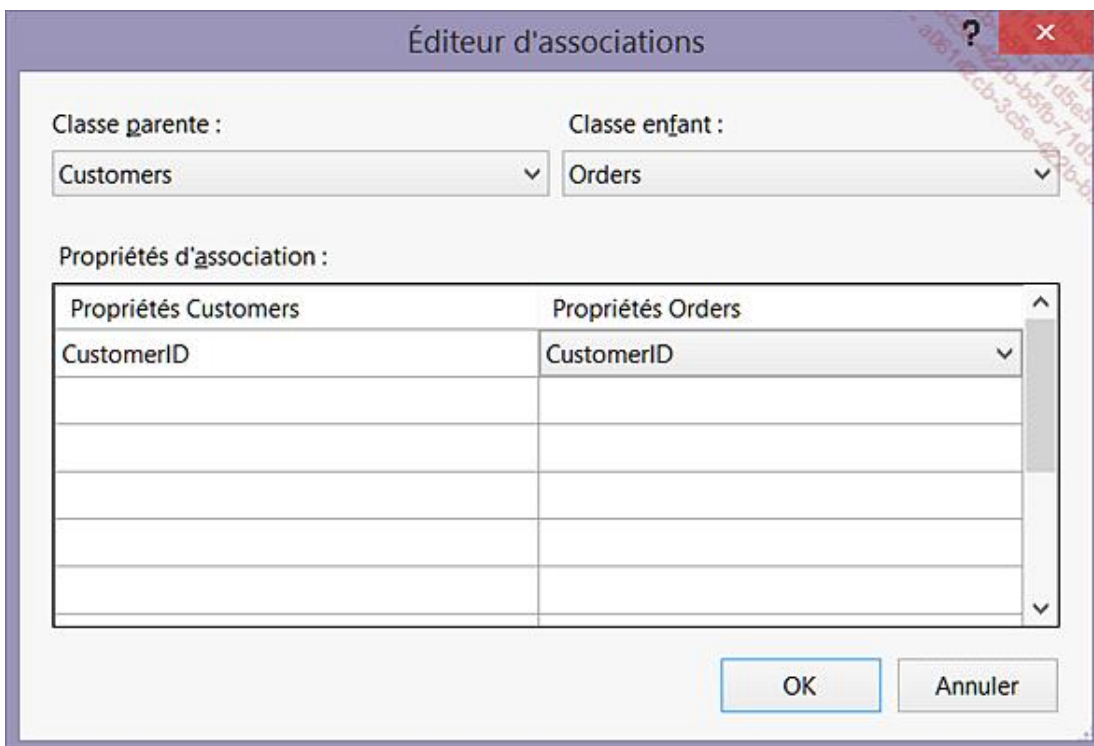
### Ajout d'association

Il est possible d'ajouter des associations dans le concepteur visuel dès lors qu'une classe est présente sur sa surface d'édition. Les associations sont l'équivalent objet des relations qui peuvent exister entre les tables dans la base de données. Lorsqu'une relation existe entre deux tables mappées, une association est d'ailleurs automatiquement ajoutée pour refléter le lien entre les tables.

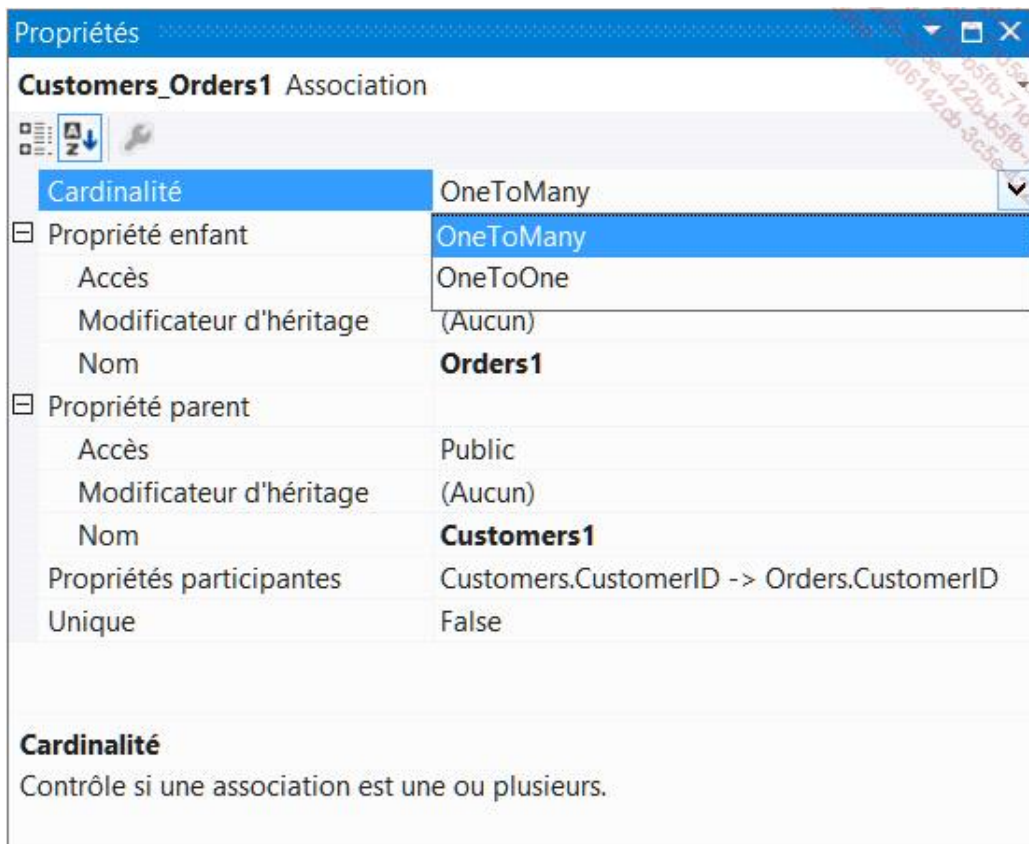
Vous pouvez ajouter une association en utilisant le menu contextuel du concepteur visuel.



La boîte de dialogue **Éditeur d'associations** permet de configurer les tables et propriétés impliquées dans l'association.



Après sa création, une association peut être éditée à l'aide de la fenêtre de propriétés de Visual Studio. Il est notamment possible de modifier la **cardinalité** de l'association pour indiquer qu'elle décrit une relation "un - un". **Propriété enfant** permet d'indiquer si une propriété de navigation doit être créée dans la classe parente. Le type de cette propriété est du type de la classe enfant si la cardinalité est OneToOne, et de type collection d'enfants si la cardinalité est OneToMany.



### Ajout de méthode

Les fonctions et procédures stockées de la base de données peuvent être mappées à l'aide du concepteur visuel. Le concepteur visuel génère pour chacune d'elles une méthode dans la classe DataContext. Ces méthodes encapsulent l'exécution de l'élément de base de données qui leur est associé. Les paramètres de la procédure stockée sont également mappés dans les paramètres de la méthode, ce qui permet de les valoriser à l'aide d'un appel de fonction C# standard.

La génération d'une méthode est effectuée par le glisser-déposer d'un de ces éléments à partir de l'explorateur de serveurs vers la surface du concepteur visuel. Il est toutefois nécessaire d'être attentif à l'emplacement de destination. Le glisser-déposer sur une classe implique que la méthode doit retourner un objet du même type. Lorsque l'élément est déplacé vers la zone droite ou vers une partie libre de la zone gauche, le type de retour de la méthode est évalué et généré automatiquement.

### Héritage

Le concepteur visuel objet/relationnel supporte la création de relations d'héritage. Cette notion est parfaitement intégrée à C#, mais est complètement absente des bases de données relationnelles. Il est donc nécessaire d'utiliser une solution de contournement pour réussir à simuler l'héritage dans une base de données.

Il est fréquent de n'utiliser qu'une seule table pour stocker les données relatives à un type parent et à ses types

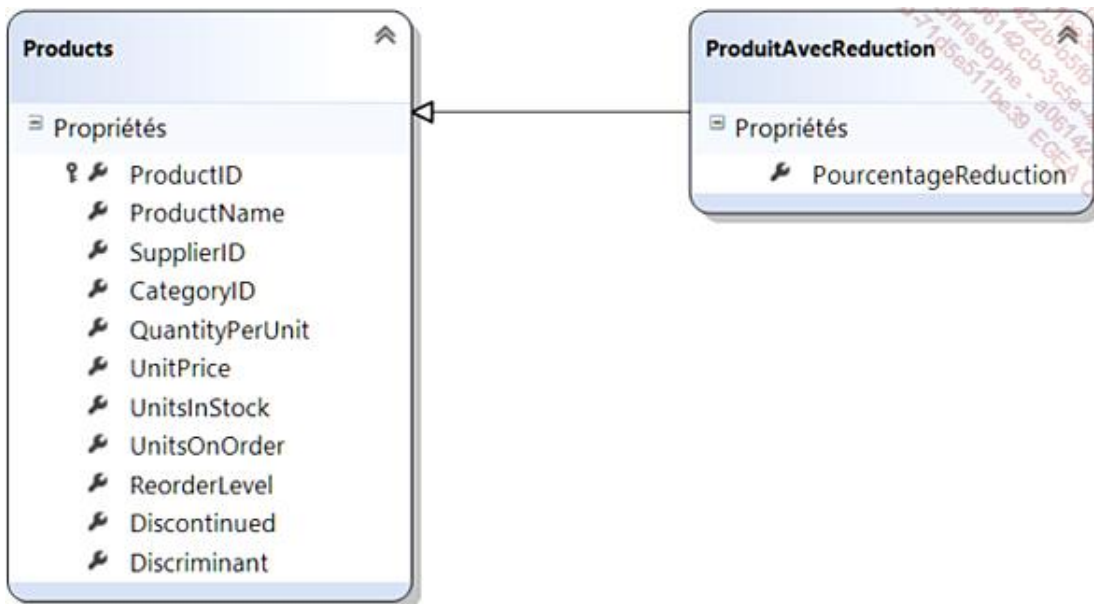
enfants. La table contient alors la totalité des colonnes liées à ces types, et une colonne supplémentaire est ajoutée. Cette colonne contient une valeur discriminante, c'est-à-dire que son contenu permet de connaître le type d'objet associé.

Avec l'utilisation de cette technique, l'ajout d'un type `ProduitAvecReduction` héritant de `Products` et possédant une propriété `PourcentageReduction` est implémenté de la manière suivante :

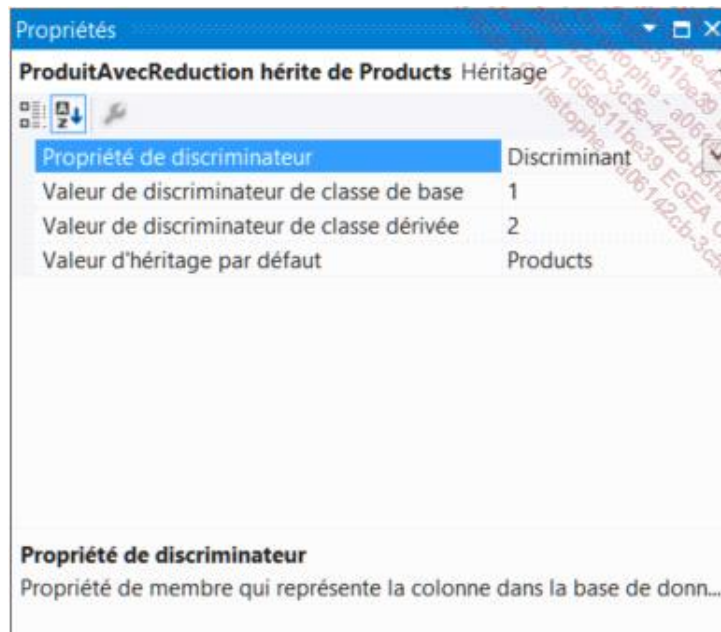
- Une colonne `Discriminant` est ajoutée à la table `Produit`. Les valeurs autorisées dans cette colonne seront 1 pour un produit et 2 pour un produit avec réduction.
- La colonne `PourcentageReduction` est ajoutée à la table `Produit`. Lorsque l'enregistrement concerne un produit, la valeur enregistrée dans cette colonne est `NULl`.

Une fois la base de données modifiée, il convient de créer le mappage correspondant.

- Ajoutez la table `Products` au modèle objet.
- Ajoutez une seconde fois la table `Products` au modèle objet et nommez-la `ProduitAvecReduction`.
- Supprimez les propriétés autres que `PourcentageReduction` dans la classe `ProduitAvecReduction`.
- Ajoutez une relation d'héritage à l'aide du menu contextuel du concepteur. Une boîte de dialogue demande la sélection de la classe de base et de la classe enfant.



- Ouvrez la fenêtre de propriétés après avoir sélectionné la relation d'héritage et indiquez les valeurs suivantes :



La valeur d'héritage par défaut permet de définir le type de l'objet créé lorsque la colonne `Discriminant` a une valeur inconnue.

## 2. Utilisation de LINQ to SQL

LINQ to SQL permet d'utiliser la syntaxe LINQ décrite dans la première partie de ce chapitre pour l'interrogation des bases de données SQL Server. La différence principale tient dans le fait que les objets manipulés sont créés à partir des données issues de la base de données.

### a. Récupération de données

La communication avec la base est gérée par le `DataContext`, ce qui implique qu'il est obligatoire d'utiliser cette classe pour accéder aux données.

Différentes collections sont exposées par cette classe. Elles correspondent aux différentes tables mappées dans le concepteur visuel. Le code définissant la propriété liée à la table `Customers` dans le `DataContext` est le suivant :

```
public System.Data.Linq.Table<Customers> Customers
{
    get
    {
        return this.GetTable<Customers>();
    }
}
```

La récupération des données de la table `Customers` est effectuée au travers d'une requête LINQ impliquant la collection associée. L'affichage de la liste des clients français est ainsi fait de la manière suivante :

```
NorthwindDataContext context = new NorthwindDataContext();
```

```

var clientsFrancais = from c in context.Customers
                        where c.Country equals "France"
                        select c;

foreach (var client in clientsFrancais)
{
    Console.WriteLine(client.CompanyName);
}

```



La classe parente de `NorthwindDataContext`, `DataContext`, implémente l'interface `IDisposable`. Il convient donc d'utiliser la structure `using` pour assurer la libération des ressources des objets de ce type.

Il est possible d'afficher la requête SQL générée par une requête LINQ en valorisant la propriété `Log` de l'objet `DataContext` avec le flux de sortie standard représenté par la console :

```
Context.Log = Console.Out;
```

Dans le cas de notre recherche des clients français, le code SQL exécuté est :

```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[Country] = @p0

```

La requête générée correspond exactement au code LINQ et aurait pu être facilement exécutable en utilisant ADO.NET. LINQ to SQL commence à dévoiler sa puissance lorsque les requêtes se complexifient. L'exemple ci-dessous affiche les dates de commandes par client.

```

using (NorthwindDataContext context = new NorthwindDataContext())
{
    var commandesClients =
        from c in context.Customers
        join comm in context.Orders on c.CustomerID equals
        comm.CustomerID into commandesClient
        select new { Client = c, Commandes = commandesClient };

    foreach (var commCli in commandesClients)
    {
        Console.WriteLine(commCli.Client.CompanyName);
        foreach (var comm in commCli.Commandes)
        {
            Console.WriteLine(comm.OrderDate);
        }
    }
}

```

Le code SQL généré est le suivant :



```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country],
[t0].[Phone], [t0].[Fax], [t1].[OrderID], [t1].[CustomerID] AS
[CustomerID2], [t1].[EmployeeID], [t1].[OrderDate],
[t1].[RequiredDate], [t1].[ShippedDate], [t1].[ShipVia],
[t1].[Freight], [t1].[ShipName], [t1].[ShipAddress],
[t1].[ShipCity], [t1].[ShipRegion], [t1].[ShipPostalCode],
[t1].[ShipCountry], (
    SELECT COUNT(*)
    FROM [dbo].[Orders] AS [t2]
    WHERE [t0].[CustomerID] = [t2].[CustomerID]
) AS [value]
FROM [dbo].[Customers] AS [t0]
LEFT OUTER JOIN [dbo].[Orders] AS [t1] ON [t0].[CustomerID] =
[t1].[CustomerID]
ORDER BY [t0].[CustomerID], [t1].[OrderID]

```

## b. Mise à jour de données

La mise à jour des données contenues dans la base est représentée par trois types d'actions : l'insertion, la modification et la suppression. Ces opérations peuvent toutes trois être exécutées à travers l'utilisation de LINQ to SQL.

### Insertion de données

L'ajout d'un enregistrement passe par la création d'un objet dont les propriétés sont valorisées. Une fois l'objet complet, il est ajouté à la table par l'intermédiaire de la collection qui lui est associée dans le DataContext. Enfin, l'appel à la méthode `SubmitChanges` du DataContext répercute cet ajout dans la base de données par la génération et l'exécution d'une requête SQL adaptée.

L'exemple ci-dessous crée un nouveau client dans la base de données.

```

Customers client = new Customers();
client.CustomerID = "JMART";
client.CompanyName = "Ma société";
client.Address = "17, grande rue";
client.City = "Toulouse";
client.ContactName = "Jean MARTIN";
client.Country = "France";
client.Phone = "0501020304";

using (NorthwindDataContext context = new NorthwindDataContext())
{
    context.Customers.InsertOnSubmit(client);
    context.SubmitChanges();
}

```

### Modification de données



LINQ permet de modifier un enregistrement en base de données d'une manière parfaitement naturelle. L'enregistrement que l'on souhaite modifier est tout d'abord récupéré sous la forme d'un objet à l'aide d'une requête LINQ. Les différentes propriétés de l'objet peuvent ensuite être revalorisées. La dernière étape consiste à appeler, comme pour l'action d'insertion, la méthode `SubmitChanges` du `DataContext`. Le code suivant effectue la modification du nom de la société dernièrement créée :

```
using (NorthwindDataContext context = new NorthwindDataContext())
{
    var client = (from c in context.Customers
                  where c.CustomerID == "JMART"
                  select c).First();

    client.CompanyName = "Ma nouvelle société";

    context.SubmitChanges();
}
```

### **Suppression de données**

La dernière action de mise à jour des données concerne la suppression d'un élément. Elle est effectuée en fournissant l'objet pour lequel l'enregistrement doit être supprimé à la méthode `DeleteOnSubmit` de la collection liée à la table concernée.

La suppression de notre client est ainsi effectuée de la manière suivante :

```
using (NorthwindDataContext context = new NorthwindDataContext())
{
    var client = (from c in context.Customers
                  where c.CustomerID == "JMART"
                  select c).First();

    context.Customers.DeleteOnSubmit(client);

    context.SubmitChanges();
}
```

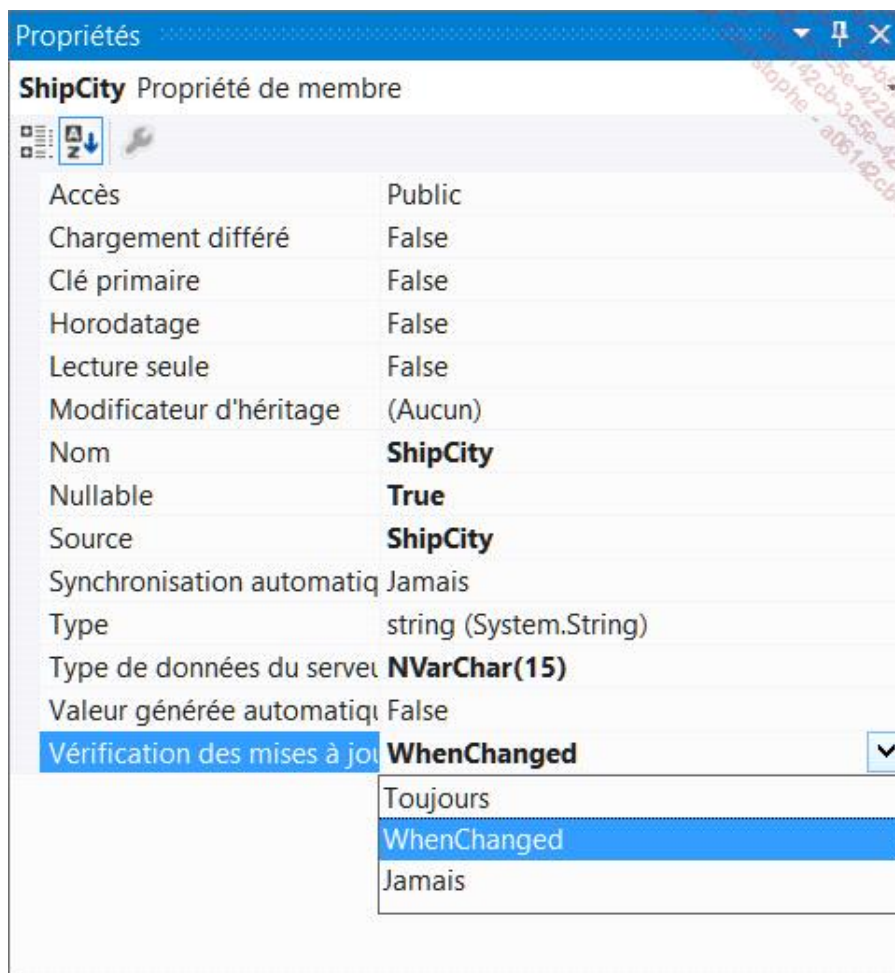
## **c. Gestion des conflits**

Comme nous l'avons vu dans le chapitre concernant ADO.NET, une des problématiques les plus importantes dans l'utilisation de bases de données est la gestion des accès concurrentiels. LINQ to SQL n'étant qu'un intermédiaire pour l'exécution de requêtes SQL, il est également possible de se trouver dans une situation de conflit pendant son utilisation. LINQ inclut un mécanisme permettant de gérer ces problèmes. Celui-ci est constitué de quatre parties complémentaires.

### **Configuration de la détection de conflits**

Lors de la création des classes de mappage avec le concepteur visuel, il est possible d'indiquer pour chaque propriété si elle est incluse dans le mécanisme de gestion des conflits, et si oui, comment sa présence dans le mécanisme doit être gérée. La fenêtre de propriétés permet d'éditer la valeur du champ **Vérification des mises à**

**jour** de manière à choisir le comportement adapté.



Les valeurs disponibles pour ce champ sont :

- **Toujours** : la détection des conflits prend toujours en compte la propriété. C'est la **valeur par défaut** pour toutes les propriétés.
- **WhenChanged** : la détection des conflits ne prend en compte cette propriété que si sa valeur a changé.
- **Jamais** : ce champ n'est jamais utilisé dans la détection des conflits.

### Détection des conflits

La détection des conflits est effectuée lorsque les données sont remontées à la base de données. Cette étape intervient donc lorsque la méthode `SubmitChanges` du `DataContext` est appelée. Celle-ci a une surcharge permettant de spécifier le mode de détection des conflits à l'aide d'une valeur de l'énumération `ConflictMode`. Deux valeurs sont disponibles :

- `FailOnFirstConflict` indique que l'exécution des mises à jour sera arrêtée dès le premier conflit. Cette valeur est celle qui est appliquée lorsque l'on utilise la méthode `SubmitChanges` ne prenant pas de paramètres.
- `ContinueOnConflict` indique que toutes les mises à jour doivent être traitées. Les éventuels conflits sont remontés à la fin de l'exécution.

La remontée des informations liées aux conflits est effectuée par le déclenchement d'une exception de type

`ChangeConflictException`. Il convient donc d'effectuer tout appel à `SubmitChanges` dans un bloc `try ... catch` afin de gérer les conflits.

### **Lecture des informations liées au conflit**

Lorsqu'un ou plusieurs conflits sont détectés, la collection `ChangeConflicts` du `DataContext` est valorisée pour fournir certaines informations.

Chaque objet de cette collection est de type `ObjectChangeConflict`. La propriété `Object` de ce type renvoie l'objet pour lequel un conflit a été détecté. La collection `MemberConflicts` permet quant à elle d'affiner l'analyse du conflit en renvoyant la liste des membres à l'origine du problème. Les objets de cette collection possèdent trois propriétés permettant d'analyser et de résoudre le problème : `OriginalValue`, `CurrentValue`, `DatabaseValue`. Elles représentent respectivement la valeur originale d'un membre, la valeur actuelle de ce membre, et enfin la valeur actuelle en base de données pour la colonne associée.

### **Résolution des conflits**

La résolution d'un conflit passe par un appel à la méthode `Resolve` de l'objet `ObjectChangeConflict` qui décrit les données problématiques. Cette méthode attend en paramètre une valeur de l'énumération `RefreshMode` :

- `OverwriteCurrentValues` : lorsque cette valeur est utilisée, les données de l'objet sont entièrement réactualisées à partir de la base de données. Les données modifiées disparaissent donc.
- `KeepCurrentValues` : cette valeur force l'écriture des valeurs actuelles en base de données.
- `KeepChanges` : les données conflictuelles de l'objet sont conservées, mais les autres propriétés sont mises à jour avec les données fraîches de la base.