



Site : www.e-naxos.com Blog : www.e-naxos.com/blog

Formation C# - Visual Studio - Delphi.NET/Win32

Audit, Conseil, Développement

© Copyright 2007 Olivier DAHAN - Version 1.1 13-12-2007

Reproduction, utilisation et diffusion interdites sans l'autorisation de l'auteur. Pour plus d'information contacter odahan@e-naxos.com

IMPORTANT : Sur certains sites le texte qui suit peut présenter une mise en page plus ou moins adaptée (par exemple s'il est transformé en html). N'oubliez pas que vous pouvez toujours **télécharger l'article original au format PDF** accompagné du **code source des exemples** sur le site e-naxos, rubrique téléchargements, groupe « articles c# ».

Pré-requis : Cet article suppose que vous ayez pris connaissance des nouveautés syntaxiques de C# 3.0. Si cela n'est pas le cas je vous suggère la lecture de mon article précédent « **les nouveautés syntaxiques de C# 3.0** » article téléchargeable sur mon site comme le présent. De plus plusieurs billets de mon blog propose des informations complémentaires sur LINQ, n'hésitez pas à les lire pour mieux cerner le sujet.

Note de la version 1.1 : Faire œuvre de vulgarisation n'est jamais chose aisée, à vouloir simplifier pour mieux faire comprendre on utilise par force quelques approximations. Dans la version 1.0 de cet article je traitais Linq to Entities mais en montrant Linq to SQL pour simplifier la démonstration. Mais sans en prévenir le lecteur cette approche avait l'inconvénient d'éventuellement brouiller un peu les idées de ce dernier entre Linq to SQL et Linq to Entities et leurs diagrammes respectifs. La version 1.1 tente de clarifier les différences entre ces deux types de modélisation. Les lecteurs y trouveront certainement mieux leur compte, en tout cas je l'espère.

Table des matières

Ce qui fait de LINQ une nouveauté	3
LINQ to Objects	4
Construction de la requête	5
LINQ to ADO.NET	7
LINQ to ADO.NET, c'est quoi ?	8
LINQ to SQL	8
LINQ to Dataset	8
LINQ to Entities	8
LINQ to XML	8
Les exemples	8
LINQ to SQL	8
Les bases du mapping	10
Un contexte d'exécution	11
Première requête	12
Variante	13
Complicquons un peu la requête	14
Filtrage	14
Tri	15
Autre syntaxe	16
Concluons sur LINQ to SQL	17
LINQ to Dataset	17
Exemple	17
LINQ to XML	19
Créer des données XML	19
Marier LINQ to XML avec LINQ to SQL/Dataset/Entities	21
LINQ to Entities	23
Développeur != Plombier	24
Changer de dimension	24
Passons à la réalité	25
Construire le modèle	27
Personnaliser le modèle	30
Premier test du modèle	31
Ajout d'une procédure stockée au modèle	32
Utiliser les relations	33
Créer des relations	34
Modifications des données	35
Conclusion	36

LINQ

Language-Integrated Query, ou Requête intégrée au langage, est un des ajouts les plus marquants du Framework 3.5 et de C# 3.0. Et les mots sont pesés.

Très raisonnablement, l'avancée conceptuelle derrière LINQ est un bouleversement énorme au moins aussi important que le Framework .NET lui-même à sa sortie avec toutes ses technologies et ses outils.

Vous aidez à prendre la mesure de ce bouleversement, c'est le but du présent article.

Une précision : Ce qui va suivre n'est pas un cours sur LINQ, plus humblement c'est un vaste tour d'horizon de cette technologie, à la fois par des explications sur le « pourquoi » et le « comment » et par des exemples de code. Je reviendrai dans d'autres articles plus en détail sur certains aspects. Pour l'instant, venez découvrir pourquoi il y aura un avant LINQ et un après LINQ et pourquoi vous refuserez de développer dans le futur avec des langages ne le supporteront pas...

Ce qui fait de LINQ une nouveauté

Si vous avez une vague idée de ce qu'est LINQ vous pensez peut-être que LINQ n'est finalement qu'un nouveau procédé d'O/R mapping (ORM) comme il en existe de longue date sous .NET ou sous Java, et même sous Delphi avec ECO.

Cette approche est fautive à plus d'un titre !

Tout d'abord LINQ n'est pas un utilitaire, un framework lourd à maîtriser, c'est une évolution naturelle de la syntaxe de C#, il fait donc partie intégrante du langage (et de la plateforme .NET qui fournit les services). Ce n'est ni un ajout, ni une verrue, ni une astuce, c'est C# 3.0, tout simplement.

Ensuite, LINQ n'est en rien un simple outil d'ORM, c'est plutôt un SQL totalement objet qui s'adapte à différents contextes, dont les bases de données mais pas seulement. Car même cette comparaison est réductrice, LINQ est bien plus simple et plus puissant par exemple que l'OCL utilisé par ECO de Borland ou que les différentes versions de Object-SQL implémentées dans certains SGBD-Orienté Objet comme OQL dans O2 par exemple. Même NHibernate prépare un NHibernate to LINQ malgré le succès de l'outil et de sa philosophie. Quant à DB4O, elle propose quelque chose de plus utilisable, d'où son relatif succès en ce moment mais tout comme la base MySQL la pub parle de gratuité, mais en réalité les licences « pros » sont onéreuses... Gardez vos sous ! Tout cela coûte plus cher qu'un indispensable abonnement à MSDN qui vous offre Visual Studio (et bien plus) avec son LINQ intégré !

LINQ est donc très différent de tout cela. Pourquoi ? Parce que tous ces langages plus ou moins intégrés à des EDI, des frameworks ou à certains SGBD-O sont totalement disjoints du langage de programmation principal utilisé par le développeur, ils ont une syntaxe propre et non cohérente avec celle du langage des applications complexifiant la conception de ces dernières et forçant une gymnastique intellectuelle à la charge du développeur pour faire le lien entre les deux mondes... On échange donc un mapping O/R pour un mapping encore plus complexe entre deux systèmes objets incompatibles ! LINQ évite tous ces problèmes, LINQ c'est du C# (ou du VB), ni plus ni moins.

LINQ est une aussi une continuité logique vers plus d'objectivation des données, mouvement entamé depuis longtemps sous .NET. On se rappellera par exemple de l'introduction dans le Framework 2.0 de l'**ObjectDataSource** permettant d'utiliser une grappe d'objets en mémoire comme source de données pour des objets d'interface sous ASP.NET, ou même du DataBinding de .NET 1.0 qui fait que tout objet avec ses propriétés peut être vu comme une source de données. Ces progrès étaient immenses (comparez le DataBinding de .NET 1.0 avec la complexité de créer un composant lié à une source de données sous VC++ et la MFC, Delphi ou VB Win32 par exemple...) mais on pouvait aller plus loin.

LINQ est, de fait, l'aboutissement de ce long voyage vers plus d'abstraction et en même temps vers plus de pragmatisme. LINQ est tout sauf compliqué, il est simple et naturel. Mais ce qu'il permet de faire est d'une incroyable puissance.

Pourquoi LINQ ? Simplement parce qu'un logiciel passe son temps à manipuler des données (au sens large) et qu'il était logique qu'un jour les langages informatiques intègrent enfin

ces dernières à leur fonctionnement. Faire de sources XML ou SQL, et même de listes d'objets, des « citoyens à part entière » du langage, c'est ça LINQ : arrêter la schizophrénie entre données du langage (entiers, chaînes, collections...) et données tout court (le plus souvent persistantes sur un SGBD mais pas seulement) en unifiant la syntaxe de manipulation quelle que soit la source.

LINQ est ainsi utilisable dans plusieurs contextes dont le développeur n'a plus à se soucier puisque LINQ permet de les manipuler de la même façon, comme des objets avec une même syntaxe. Et c'est en cela que LINQ est une réelle innovation. Au même titre que le Framework lui-même permet avec un même EDI, un même langage, une même librairie, de concevoir aussi bien des applications Windows que des applications Web ou des applications pour Smartphone, LINQ avec la même syntaxe permet de trier, de filtrer et de mettre à jour des données qu'il s'agisse d'objets métiers, de fichiers XML ou de sources SQL.

LINQ se décline en plusieurs « arômes » partageant tous une syntaxe de base et l'esprit de LINQ. Mais chacun est spécialisé pour un contexte donné. Pour comprendre la justification et le fonctionnement de ces différentes adaptations de LINQ, suivez le guide...

LINQ to Objects

LINQ to Objects pourrait être vu comme le socle de base. Il a pour but de permettre l'interrogation et la manipulation de collections d'objets. Ces dernières pouvant avoir des *relations hiérarchiques* avec d'autres collections et ainsi former un *graphe*.

Par essence LINQ est donc relationnel, mais pas au sens des bases de données, au sens d'un langage objet.

Les bases de données SQL sont fondées sur la notion de tableaux appelés tables, c'est-à-dire des surfaces rectangulaires accessibles en ligne / colonne. Tout est table sous SQL, et un résultat de requête est aussi une table. C'est là toute la puissance de SQL mais c'est aussi sa faiblesse : il ne peut pas retourner un graphe, juste un rectangle avec des cases. LINQ travaille sur des graphes et sait retourner des graphes. SQL ne sait tout simplement pas le faire.

D'ailleurs LINQ to Objects ne se limite pas à la manipulation des objets créés par le développeur, il permet d'interroger n'importe quelles données, même celles retournées par le système du moment qu'il s'agit d'une collection. Pour s'en convaincre et appréhender tout de suite l'intérêt évident de LINQ prenons un exemple très simple (peu importe pour le moment que la syntaxe vous échappe peut-être) qui consiste à lister les fichiers du répertoire temporaire de Windows.

```

public static void LanceDemo()
{
    string temp = Path.GetTempPath();
    DirectoryInfo info = new DirectoryInfo(temp);
    var query = from f in info.GetFiles()
                where f.Length > 1024 * 10
                orderby f.Length descending
                select new { f.Length, f.LastWriteTime, f.Name };

    ObjectDumper.Write(query);
    Console.ReadLine();
}

```

La sortie à la console, lorsqu'on appelle la méthode **LanceDemo()** est la suivante :

Length=57689462	LastWriteTime=20/11/2007	Name=VSMsiLog26AC.txt
Length=56904858	LastWriteTime=06/08/2007	Name=VSMsiLog6E4B.txt
Length=19757712	LastWriteTime=19/11/2007	Name=VSMsiLog3BA0.txt
Length=18874368	LastWriteTime=06/08/2007	Name=WinSAT_DX.etl
Length=7340032	LastWriteTime=06/08/2007	Name=WinSAT_KernelLog.etl
Length=5891268	LastWriteTime=20/11/2007	Name=dd_WMPPC_5_0_MSI31C5.txt
Length=5882250	LastWriteTime=06/08/2007	Name=dd_WMPPC_5_0_MSI772C.txt
...		
Length=11436	LastWriteTime=07/08/2007	Name=SilverlightUI403E.txt
Length=11436	LastWriteTime=06/08/2007	Name=SilverlightUI77FC.txt
Length=11378	LastWriteTime=13/11/2007	Name=SilverlightUI11EE.txt
Length=10996	LastWriteTime=08/11/2007	Name=vs_setup.pdi

Lorsque la méthode **LanceDemo()** est invoquée, elle affiche à la console tous les fichiers du chemin temporaire de l'OS en ne prenant que ceux dont la taille est supérieure à 10 Ko, le tout en présentant les fichiers par ordre décroissant de taille. Seule trois informations par fichier sont retournées : sa longueur, sa date de dernière écriture et son nom. Êtes-vous capable de faire un tel traitement aussi clairement et en si peu de lignes sans LINQ ? ... Vous connaissez la réponse, c'est non. Et vous commencez donc à comprendre tout l'intérêt de LINQ même avec de simples liste d'objets (et je ne parle pas des graphes d'objets !)...

L'affichage est réalisé par ObjectDumper qui est une petite classe outil réalisant tout simplement un foreach pour lister à la console toute collection qui lui est passée. La classe utilise la réflexion pour afficher le nom des propriétés avec leur valeur. Elle n'a rien à voir avec LINQ, c'est une astuce pour afficher rapidement des données. Le code source, provenant de Microsoft, est fourni avec le code des exemples de l'article.

Construction de la requête

En dehors de l'objet utilitaire d'affichage que vous ne connaissez pas et qui n'a pas d'intérêt dans nos explications, le reste est d'une grande lisibilité : nous obtenons le chemin des fichiers temporaires de Windows puis nous construisons une requête LINQ.

LINQ ressemble beaucoup ici à SQL mais avec certaines spécificités. La première chose et la plus importante c'est qu'il s'agit de code C#, pas d'une chaîne de caractères ne voulant rien dire pour le langage. Il est ainsi contrôlé à la compilation et on dispose d'Intellisense lors de son écriture. Bien entendu, vous noterez aussi que le plus puissant des SQL ne

pourrait rien dans ce cas précis puisque la source est une liste renvoyée par l'OS via le Framework .NET.

Ensuite vous remarquez que par rapport à SQL la requête est en quelque sorte « inversée ». La clause **from** est placée en premier et la clause **select** en dernier...

Pourquoi ce choix qui peut sembler curieux de prime abord ?

L'explication m'a été donnée par Luca Bolognese dans sa conférence sur LINQ aux TechEd 2007 à Barcelone (conférence TLA 308 pour ceux qui y ont participé et qui souhaiterait la voir ou la revoir sur le DVD) : Visual Studio possède une fonction essentielle pour accélérer le développement, *Intellisense*. Et pour que ce dernier puisse fonctionner correctement il faut qu'il sache sur quoi on travaille, ce qui est le cas le plus souvent. Mais dans une requête SQL (ou de type SQL) si on commence par taper le **select** Intellisense sera dans l'impossibilité de fournir une aide concernant les champs disponibles puisque le développeur n'a pas encore tapé la clause **from**... Donc pour permettre à Intellisense d'être utilisable même lors de la frappe des requêtes LINQ il fallait que le nom de la ou des sources de données, le **from**, se trouve en premier. C'est tout bête.

De plus, Luca parle aussi d'expériences menées par Microsoft où des développeurs purement SQL étaient placés devant des écrans pour taper des requêtes, le tout sous l'œil indiscret d'observateurs professionnels cachés derrière des vitres sans tain. Lors de ces expériences il a pu être noté que le plus souvent un développeur SQL commence par taper la clause **from** pour savoir sur quoi porte sa requête puis qu'il « remonte » pour taper le **select**. LINQ ne ferait donc que reprendre l'ordre réel dans lequel les développeurs saisissent du SQL, même sans en avoir conscience...

Bref, LINQ commence par la clause **from** dans laquelle on nomme une source (et aussi d'éventuelles jointures comme nous le verrons ultérieurement), dans notre exemple la source est fournie par la collection retournée par **GetFiles()** de l'objet **DirectoryInfo**. Dans cette requête on décide d'appeler chaque élément de cette source **f**, au même titre que dans un **foreach** on nomme la variable qui sera renseignée à chaque passage dans la boucle. Il ne faut pas confondre cette notation qui ressemble au premier coup d'œil à un alias de table en SQL (**FROM Customer CLIENTS**). Sous SQL il s'agit bien de l'alias de l'ensemble de données (la table), alors qu'ici il s'agit d'un élément de la collection. A ce stade notre requête possède donc une source de données (la liste des fichiers du répertoire temporaire de l'OS), source qui alimentera à chaque passage une variable appelée **f**. Le type de **f** est déduit de celui de la collection automatiquement (inférence de type comme par le mot clé **var**, voir mon précédent article sur les nouveautés de C# 3.0).

On retrouve ensuite des mots clés classiques de SQL comme la clause **where**. Ici elle nous sert à filtrer sur la taille des fichiers retournés. La clause **orderby** fonctionne comme son homonyme SQL, elle trie les données retournées, ici en ordre descendant avec l'ajout de **descending**.

La clause **select** de l'exemple est peut-être celle qui sera la moins simple à comprendre de prime abord même si son nom est familier en SQL.

En réalité, dans la version la plus simple nous aurions pu écrire `select f` ; c'est-à-dire retourne l'élément `f` de la collection qui répond aux critères de la requête. Dans ce cas la variable `query`, qui contient (virtuellement) le résultat de la requête aurait été une collection d'objets `FileInfo`.

Mais nous ne sommes pas intéressés ici par toutes les informations de `FileInfo`, seuls le nom du fichier, sa taille et sa date de dernière écriture sont utiles pour cette application (choix arbitraire pour la démo, bien sûr). Pourquoi se charger d'objets plus lourds si notre besoin est plus léger ? ... LINQ permet de répondre à cette question très facilement : par le mot clé `new` qui va permettre de retourner une instance de tout type et donc, aussi, un type anonyme (*voir mon article sur les nouveautés de C# 3.0*), ce type anonyme, cette nouvelle classe sans nom, sera constituée de trois champs, les trois que nous listons. Nous ne leur donnons pas de nom car LINQ est assez fin pour les déduire de ceux des propriétés de `f` (une instance de `FileInfo`) ... N'indiquant aucun nom, LINQ reprend automatiquement ceux des propriétés de l'objet utilisé (ceux de la classe `FileInfo` donc).

Voilà ce qu'est LINQ to Objects et voilà par un exemple on ne peut plus simple pourquoi cette nouveauté de C# 3.0 va devenir aussi indispensable que tout le reste du langage : tout est données, même une liste de fichiers retournée par l'OS et cette liste peut être filtrée, triée en quelques mots en bénéficiant de Intellisense...

Un dernier mot. Un peu plus haut je parle de la variable `query` (déclarée par `var`) et j'ajoutais qu'elle contient le résultat de la requête mais « virtuellement ». Pourquoi cette nuance ?

Simplement parce les requêtes LINQ ne sont exécutées qu'au moment où leur résultat est énuméré. Dans notre code cela a lieu dans la boucle de l'objet `ObjectDumper`. Le fait d'écrire une requête LINQ ne déclenche rien. On peut ainsi les regrouper à un même endroit pour plus de lisibilité ou d'autres raisons, elles ne déclencheront leur traitement qu'au moment où on tentera d'énumérer les éléments retournés.

LINQ to ADO.NET

Nous venons de voir que LINQ to Objects est une fonctionnalité de base de C# 3.0 qu'on peut utiliser sur toute collection d'objets (ou graphe d'objets). Nous n'avons utilisé qu'une liste simple dans l'exemple précédent, mais LINQ fonctionne sur des graphes (avec des jointures), c'est-à-dire sur des propriétés qui retournent d'autres collections. Par exemple la fiche d'un client possèdera une propriété « commandes » qui est une collection de toutes les commandes de ce client qui elles-mêmes ont une propriété « lignes » retournant les lignes de chaque commande, etc. LINQ autorisera la navigation dans telles propriétés et sous-propriétés, comme on navigue sur les relations d'une base de données.

L'exemple de LINQ to Objects ne montrait pas cette possibilité car nous allons avoir d'autres occasions de mettre en œuvre la gestion de vrais graphes d'objets, notamment avec LINQ to ADO.NET. Gardez seulement à l'esprit que cela s'applique aussi à LINQ to Objects.

LINQ to ADO.NET, c'est quoi ?

On appelle LINQ to ADO.NET l'ensemble des implémentations de LINQ qui, d'une façon ou d'une autre, doivent accéder à des données relationnelles accessible via les fournisseurs d'accès de ADO.NET.

LINQ to ADO.NET regroupe ainsi trois saveurs différentes de « LINQ »:

LINQ to SQL

Cette version de LINQ gère le mapping entre des types C# et les enregistrements d'une base de données. C'est en quelque sorte le niveau d'entrée de LINQ to ADO.NET.

LINQ to Dataset

Il s'agit de la possibilité d'interroger avec LINQ les données contenues dans un Dataset.

LINQ to Entities

Cette version de LINQ possède un niveau d'abstraction supérieur à LINQ to SQL. Au lieu d'interroger la base de données et d'avoir une sorte d'équivalence entre classes manipulées par LINQ et tables ou vues présentes dans la base de données, LINQ to Entities permet d'écrire des requêtes qui portent sur un *modèle conceptuel*, modèle lui-même mappé sur la base physique. Nous en verrons aussi un exemple car il s'agit de la version de LINQ la plus sophistiquée et la plus puissante.

LINQ to XML

Cette version de LINQ a été adaptée pour permettre la manipulation de données XML. On retrouve ici toute la justification et toute la puissance de LINQ au service des données XML, de plus en plus présentes mais toujours aussi lourdes à exploiter. Grâce à LINQ to XML, et par l'ajout de quelques spécificités permettant de gérer les balises XML, il devient possible d'écrire des requêtes complexes sur des sources XML en quelques instructions et même de construire des fichiers XML de façon intuitive. Loin d'être anecdotique, LINQ to XML révolutionne lui aussi par ses ajouts syntaxiques la façon de travailler avec des données XML.

Les exemples

Le but de cet article est de vous fournir une vision d'ensemble de LINQ et pour cela, plutôt que de rester dans la théorie je souhaite vous montrer du code au travers d'exemples concrets et parlants. En revanche je ne traiterai pas en détail de la syntaxe utilisée, la documentation de Microsoft me semble suffisamment claire pour ne pas en faire une redite (et je suis certain que vous comprendrez aussi qu'un article ne peut se transformer en un livre !).

LINQ to SQL

Si LINQ to Objects est déjà en soi un grand progrès, avec LINQ to SQL on franchit encore une étape qui marquera un tournant dans les langages de programmation, j'en suis convaincu.

En effet, depuis des décennies les développeurs passent une bonne partie de leur temps à tenter de marier deux mondes totalement disjoints et s'éloignant avec le temps : d'une

part les langages, d'abord procéduraux, puis simplement « orientés objet » jusqu'au langage comme C# totalement objet, et, d'autre part, les bases de données, d'abord simples fichiers indexés, puis base de données relationnelles de plus en plus sophistiquées. Deux mondes qui ont connu un essor aussi important mais dans deux directions qui ne se sont jamais rejointes.

Les bases de données, comme je le soulignais en début d'article, ne savent travailler que sur un modèle ligne / colonne, avec une entité de base, la table (appelée aussi relation). C'est même leur fondement, la toute première des 12 règles du docteur Codd qui fonda le « modèle relationnel » dans son papier « *A Relational Model of Data for Large Shared Data Banks* » en 1970 (la règle n°1 impose que toute l'information soit présentée uniquement sous la forme de lignes et de colonnes - une table).

Quelles que soient les évolutions des SGBD, l'ajout des champs Blobs, le support des données texte longues, des données XML, voire des « objets », tout cela reste régi par les 12 règles de Codd et ne peut s'en écarter au risque de faire écrouler l'édifice technologique et industriel des SGDB relationnels. D'où la naissance des SGBD objet. Mais si le monde des SGBD relationnels est bien balisé, reposant sur des technologies éprouvées, il n'en va pas de même des SGBD-O. Le premier frein est leur lenteur légendaire, le second est que par manque de standard imposé par les faits, chacune possède un langage d'interrogation plus ou moins ésotérique (API complexes ou variantes exotiques de OQL généralement). Pire, les objets de ces SGBD-O n'ont que très peu de chance de correspondre aux objets des langages utilisés par les développeurs ce qui réclament à nouveau un système de traduction et de mapping... On ne gagne donc pas grand-chose et on se complique surtout la vie (et la maintenance !).

On constate ainsi que malgré quelques tentatives intéressantes les SGBD-O ne rencontrent le plus souvent qu'un succès d'estime comme en ce moment DB4O ou même O2.

Bref, les SGBD-R ont encore de très beaux jours devant eux car ils sont standardisés, rapides, puissants et de nombreux acteurs de ce marché sont implantés depuis suffisamment longtemps pour que chacun puisse faire un choix éclairé et bénéficier de produits rapides, puissants et fiables. SQL Serveur, MySQL, Sybase, DB2 ou Oracle sont des exemples très factuels de ces avantages.

Ce constat serait-il celui d'un statu quo, d'un mariage définitivement impossible entre langage de programmation Objet et SGBD-R ? Comment imaginer marier ces bases de données relationnelles coincées dans leur logique en deux dimensions, ne sachant travailler que sur des rectangles, ne sachant retourner que des rectangles, avec des langages objets eux aussi de plus en plus sophistiqués et travaillant sur des graphes ?

C'est le O/R mapping (ou ORM, *object-relational mapping*), c'est-à-dire un middleware à qui on apprend à la fois comment les données sont stockées dans la base de données et comment elles sont représentées en mémoire par des objets du langage de programmation. A sa charge de créer ces derniers à partir des informations contenu dans la base. L'ORM est une technique intéressante à plus d'un titre puisqu'elle permet au développeur de manipuler des objets en conformité avec la puissance des langages modernes tout en exploitant, pour la persistance, des SGBD-R fiables et largement répandus dans les entreprises.

Le seul problème de l'ORM c'est qu'il n'existe là non plus aucun vrai standard industriel ni même technique, puisque cela va d'une simple couche de type Data Access Layer (DAL) à des Frameworks entiers disposant de leur propre langage d'interrogation, bref, on trouve de tout. Faire un choix est chose ardue. Et quand un développeur ne sait pas comment choisir, il ne choisit pas : il évite et contourne le problème...

Le Framework .NET nous permettait déjà d'écrire des couches d'accès aux données objectivées, les DAL évoqués ci-dessus. Il s'agit d'ailleurs plus d'une sorte de *design pattern* que d'une innovation de la plateforme, c'est une façon d'isoler le code de la source de données très efficace et élégante utilisable, et utilisée, sous d'autres langages.

Pour aller encore plus loin le Framework .NET a proposé les Datasets typés, le mapping est automatisé, mais l'objet final, le Dataset n'est qu'une objectivation grossière de la source, il continue de présenter des données rectangulaires, pas des graphes d'objets, faisant « remonter » telle une bulle les concepts et la rigidité des SGBD au niveau des objets et du langage. On oblige toujours le développeur et son langage Objet à se plier à la logique des rectangles, le progrès est purement stylistique et pratique : moins de code à saisir (génération automatique de Visual Studio), collections de lignes de données plus simples à manipuler que des requêtes SQL individuelles, typage des informations.

Il fallait donc aller plus loin. Il fallait résoudre le mapping à sa plus simple expression, éviter les redites dans le code, unifier les concepts et surtout les rendre accessibles naturellement, en faire une extension du langage et de la plateforme. LINQ to ADO.NET c'est cela. LINQ to SQL n'est que le premier niveau, celui que nous allons voir maintenant. LINQ to Entities va encore plus loin pour atteindre le niveau d'abstraction le plus élevé possible.

Commençons par le début donc, voici LINQ to SQL.

Dans sa version la plus simple, LINQ to SQL permet de mapper n'importe quelle classe à n'importe quelle table d'une base de données. Nous utiliserons la célèbre base *Northwind* qui, si vous ne la connaissez pas encore, est une base de données classique de type clients / commandes / articles fournie avec SQL Server et utilisée systématiquement dans les démos, rituel auquel je vais faire allégeance...

Les bases du mapping

Je vous propose dans un premier temps de créer une classe **Client** toute simple :

```
public class Client
{
    public string CustomerID { get; set; }
    public string ContactName { get; set; }
    public string Country { get; set; }
}
```

Difficile de faire plus simple... La classe expose trois propriétés, l'identificateur du client, le nom du contact et le pays du client.

*Nota : les propriétés sont ici déclarées en utilisant une nouveauté de C# 3.0. Cette notation ressemble comme deux gouttes d'eau à celle d'une interface. Les accesseurs (**get** et **set**) sont indiqués mais sans code. La ressemblance est*

frappante mais elle s'arrête là. Nous définissons bien une propriété « normale », sauf que C# créera lui-même le champ privé correspondant (on peut voir les noms de ces champs dans le code IL après compilation). C'est une façon rapide et pratique de créer des propriétés sans, au départ, implémenter de champ privé ou d'accesseurs complexes, tout en laissant la possibilité de faire évoluer le code ultérieurement.

Comment mapper cette classe sur la table **Customers** de Northwind ?

Il suffit d'utiliser des attributs disponibles par **System.Data.Linq**.

Le premier attribut s'applique à toute la classe et permet d'indiquer à quelle table elle correspond. Le second attribut s'applique à chaque propriété qui possède un pendant dans les colonnes de cette table. Le code de notre classe devient alors :

```
[Table(Name="Customers")]
public class Client
{
    [Column(IsPrimaryKey = true)]
    public string CustomerID { get; set; }
    [Column]
    public string ContactName { get; set; }
    [Column]
    public string Country { get; set; }
}
```

On peut voir ci-dessus l'attribut **Table** ainsi que l'attribut **Column**. Ces attributs supportent des propriétés ou des constructeurs personnalisés pour affiner leur signification. Ainsi l'attribut **Table** possède une propriété **Name** qui permet d'indiquer le nom physique de la table dans la base de données puisque notre classe possède un nom très différent.

De même l'attribut **Column** supporte, entre autre, la propriété **IsPrimaryKey** pour indiquer que la propriété est la clé primaire de la table, permettant à LINQ de gérer cette contrainte. Comme dans cet exemple j'ai utilisé des noms de propriétés identiques aux noms des colonnes de la table il n'y a rien à ajouter. On pourrait bien entendu faire comme pour la table et préciser le nom des champs dans l'attribut **Column**. Tout cela reste donc simple tout en offrant une certaine souplesse.

Un contexte d'exécution

Pour exploiter notre classe **Client**, nous devons disposer d'un contexte faisant le lien entre cette dernière et la base de données, pour ce faire nous allons dériver notre propre classe de **System.Data.Linq.DataContext** en y ajoutant une propriété pour la table des clients. Cela s'effectue en deux lignes de code :

```
public class DemoContext : DataContext
{
    public Table<Client> Customers;
    public DemoContext(string fileOrServerOrConnection)
        :base(fileOrServerOrConnection) {}
}
```

En créant notre propre classe dérivée de **DataContext** nous pourrons très facilement accéder à la table des clients qui en devient une propriété (**Customers**).

Nous verrons qu'il existe une autre façon de faire qui évite même d'avoir à dériver **DataContext**.

Une fois cela écrit, c'est fort peu vous en conviendrez, nous pouvons travailler sur des objets **Client** provenant de la base de données sans aucun « contact » avec le monde des SGBD !

Première requête

```
public static void LanceDemo()
{
    const string northWind = "chaîne de connexion ado.net";
    DemoContext db = new DemoContext(northWind);
    db.Log = Console.Out;           // affiche le SQL sur la console
    var query = db.Customers;      // requête ultra simple: toute la table
    ObjectDumper.Write(query);     // affiche le résultat
    Console.ReadLine();           // pause clavier
}
```

Voici un extrait de la sortie console que produit ce code :

```
SELECT [t0].[CustomerID], [t0].[ContactName], [t0].[Country]
FROM [Customers] AS [t0]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

CustomerID=ALFKI      ContactName=Maria Anders      Country=Germany
CustomerID=ANATR      ContactName=Ana Trujillo      Country=Mexico
CustomerID=ANTON      ContactName=Antonio Moreno    Country=Mexico
CustomerID=AROUT      ContactName=Thomas Hardy      Country=UK
CustomerID=BERGS      ContactName=Christina Berglund Country=Sweden
CustomerID=BLAUS      ContactName=Hanna Moos        Country=Germany
CustomerID=BLONP      ContactName=Frédérique Citeaux Country=France
```

Dans la capture ci-dessus on remarque le texte de la requête SQL réellement envoyée à base de données, c'est une astuce pratique de débogage sur laquelle je vais revenir plus bas. Restons concentrés sur la liste qui suit : on obtient bien la liste de tous les clients de la table **Customers** de la base de données Northwind, chaque fiche possédant bien les trois seules propriétés de la classe **Client** que nous avons vu plus haut. Et c'est tout !

Peut-on rêver ORM plus simple et intégré plus naturellement au langage ?

Certes, pour l'instant nous restons accrochés au modèle physique avec un mapping posant une équivalence entre une classe C# et une table de la base de données. C'est LINQ to SQL. Pour atteindre le niveau supérieur nous devons utiliser LINQ to Entities. Mais pas de précipitation !

Revenons sur le code de l'exemple.

Tout d'abord vous remarquerez la constante qui définit la chaîne de connexion ADO.NET vers la base de données. Cette chaîne peut aussi être issue du fichier de paramétrage de l'application ou de l'utilisateur, bien entendu.

Ensuite nous créons une instance de notre **DataContext** personnalisé, **DemoContext**, auquel nous passons la chaîne de connexion, la variable résultante est **db**.

Petite astuce au passage, comme nous voulons voir pour les besoins de la démo le code SQL qui est réellement envoyé à la base de données, nous faisons pointer la propriété **Log** du **DataContext** vers le flux de sortie de la console. Simple et pratique. Nous verrons plus loin à quel point les requêtes LINQ, tout en restant simples, peuvent générer un SQL bien formé mais complexe que nous n'avons pas, heureusement et grâce à LINQ, à écrire !

La requête que nous utilisons dans cet exemple est la plus simple qu'on puisse concevoir : nous demandons la totalité de la table des clients.

Plus de lignes et de colonnes, plus de tables, plus de données rectangulaires, rien que des instances de la classe **Client**, c'est LINQ qui s'occupe de discuter avec le SGBD dont nous n'avons plus rien à savoir. Plus de couche D.A.L. (*Data Access Layer*) non plus, ni même éventuellement de B.O.L. (*Business Object Layer*) puisque la classe **Client** remplit le rôle de D.A.L. et qu'en y ajoutant un peu de code elle pourrait devenir dans des cas simples en même temps un B.O.L. !

Variante

L'écriture de la classe **DemoContext** dérivée de **DataContext** n'est certes pas bien compliquée... deux lignes de code très utiles puisque notre classe sait ensuite fournir la liste des tables qu'elles connaît (nous aurions pu ajouter de la même façon que les clients, les articles, les commandes etc.).

Mais il est possible de se passer de cette écriture en utilisant les génériques. Dans ce cas, le seul code à concevoir est celui de la classe **Client**... A l'exécution notre code devient le suivant :

```
public static void LanceDemo2()
{
    const string northWind = "chaîne de connexion";
    DataContext db = new DataContext(northWind);
    Table<Client> Clients = db.GetTable<Client>();
    db.Log = Console.Out;
    var query = Clients;
    ObjectDumper.Write(query);
    Console.ReadLine();
}
```

Dans cette version nous créons directement une instance de **DataContext** sans avoir dérivé cette classe. Bien entendu, **DataContext** ne sait rien de notre table des clients, il n'est donc plus possible de l'utiliser pour obtenir cette dernière. Nous créons ainsi à la volée une variable **Clients** de type **Table<Client>** dont le contenu est récupéré par **GetTable<Client>()**;

La requête est ensuite la même. En réalité elle devient superflue... La variable **Clients** contient déjà (potentiellement) tous les clients. Nous pouvons supprimer la variable **query** et passer directement la variable **Clients** à l'objet afficheur :

```
public static void LanceDemo2()
{
    const string northWind = "Data Source=\\sqlexpress;Initial
Catalog=Northwind;Integrated Security=True";
    DataContext db = new DataContext(northWind);
    db.Log = Console.Out;
    Table<Client> Clients = db.GetTable<Client>();
    ObjectDumper.Write(Clients);
    Console.ReadLine();
}
```

Vous pourrez d'ailleurs vérifier sur la console que le **DataContext** génère toujours la même requête SQL.

Dans cette version simplifiée nous n'avons fait que décrire la classe **Client** avec deux attributs (**Table** et **Column**). Rien d'autre. Et il nous est possible d'obtenir des clients, de les filtrer, de les trier mais aussi de les modifier. Nous verrons cela un peu plus loin.

Compliquons un peu la requête

Dans l'exemple qui précède nous avons tellement peu utilisé le requêtage de LINQ to SQL que la dernière version du code pouvait même se passer d'une variable pour la requête. Il était intéressant de voir à l'œuvre les mécanismes de base de LINQ, mais le temps est venu d'écrire de vraies requêtes comme nous le ferions en SQL, mais sur des objets avec LINQ.

Nous reprendrons la même structure de code et le même objet **DemoContext** dérivé de **DataContext** exposant la propriété **Customers**.

Filtrage

La première chose qu'on demande à une requête est bien souvent de filtrer les données. Essayons d'obtenir uniquement les clients situés à Londres :

```
public static void LanceDemo3()
{
    DemoContext db = new DemoContext(northWind);
    db.Log = Console.Out;
    var query = from c in db.Customers
                where c.Ville == "London"
                select c;
    ObjectDumper.Write(query);
    Console.ReadLine();
}
```

Le principe restant le même nous nous concentrerons ici uniquement sur la requête. Elle reste très simple et compréhensible pour qui connaît déjà SQL, et même pour les autres d'ailleurs, pour peu qu'on dispose d'un minimum de vocabulaire anglais.

Si nous regardons la trace SQL (**db.Log**) la requête envoyée à la base de données devient désormais :

```
SELECT [t0].[CustomerID], [t0].[ContactName], [t0].[Country], [t0].[City] AS  
[Ville]  
FROM [Customers] AS [t0]  
WHERE [t0].[City] = @p0  
-- @p0: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [London]  
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

On remarque que LINQ to SQL utilise systématiquement des paramètres, ici **@p0**. On peut aussi voir la liste des paramètres et de leurs valeurs sous la requête. La trace **Log** du **DataContext** est très instructive pour qui veut vérifier le SQL transmis à la base de données.

En effet, outre l'intérêt pédagogique que nous exploitons ici, il peut s'avérer dans certains cas nécessaire de proposer sa propre requête, voire le nom d'une procédure stockée pour atteindre les données. LINQ to SQL le gère aussi. Et pour décider quel code sera le plus efficace il s'avère indispensable d'obtenir le code SQL produit par LINQ pour l'inspecter, le **Log** répond à ce besoin.

Tri

Trier des données est tout aussi fréquent que de les filtrer. Voici un exemple de la même requête avec un tri :

```
public static void LanceDemo4()  
{  
    DemoContext db = new DemoContext(northWind);  
    db.Log = Console.Out;  
    var query = from c in db.Customers  
                where c.Ville == "London"  
                orderby c.ContactName ascending  
                select new { c.ContactName, c.Ville };  
    ObjectDumper.Write(query);  
    Console.ReadLine();  
}
```

Ici nous trions les clients retournés par ordre ascendant du nom du contact. Au passage nous ne retournons pas des instances de **Client** mais un type anonyme ne contenant que deux propriétés, le nom du contact et le nom de la ville.

La trace complète à la console donne ceci :

```

SELECT [t0].[ContactName], [t0].[City] AS [Ville]
FROM [Customers] AS [t0]
WHERE [t0].[City] = @p0
ORDER BY [t0].[ContactName]
-- @p0: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [London]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

ContactName=Ann Devon      Ville=London
ContactName=Elizabeth Brown      Ville=London
ContactName=Hari Kumar      Ville=London
ContactName=Simon Crowther      Ville=London
ContactName=Thomas Hardy      Ville=London
ContactName=Victoria Ashworth      Ville=London

```

Autre syntaxe

Arrivé à ce stade de l'article je pense que vous commencez à comprendre comment marche LINQ. Mais en réalité ce que nous avons vu n'est qu'une toute petite partie de la syntaxe de LINQ et donc de ses possibilités... Rappelons que ce que vous êtes en train de lire est juste un article présentant LINQ et non un cours sur LINQ. Je vous encourage vivement à consulter la documentation de Microsoft. Et comme je suis taquin et que je ne voudrais surtout pas vous laisser croire qu'il n'y a pas grand-chose d'autre à voir, je vais vous présenter une autre requête LINQ to SQL que je n'expliquerais pas, juste pour créer un peu de frustration et vous donner l'envie d'aller chercher au-delà de cet article...

```

public static void LanceDemo5()
{
    DemoContext db = new DemoContext(northWind);
    // db.Log = Console.Out;
    var query = db.Customers.GroupBy(c => c.Ville,
                                    c => c.Ville+": "+c.ContactName).
        Skip(10).Take(8);
    ObjectDumper.Write(query,1);
    Console.ReadLine();
}

```

Quelques indices malgré tout : la requête prend les 8 premiers clients qui suivent les 10 premiers de la table, une fois celle-ci groupée sur la ville de chaque client. Vous n'avez pas tout compris ? ... Je vous l'ai dit, c'est fait exprès !

La trace de sortie est :

```

...
Bräcke: Maria Larsson
...
Brandenburg: Philip Cramer
...
Bruxelles: Catherine Dewey
...
Buenos Aires: Patricio Simpson
Buenos Aires: Yvonne Moncada
Buenos Aires: Sergio Gutiérrez
...
Butte: Liu Wong
...
Campinas: André Fonseca
...

```


Caracas: Manuel Pereira

...

Charleroi: Pascale Cartrain

Vous retrouverez bien entendu le code de tous les exemples de cet article dans les projets VS 2008 fournis dans le même fichier Zip que le PDF que vous lisez en ce moment.

Concluons sur LINQ to SQL

L'intention de cet article n'est pas, comme je l'ai déjà mentionné, d'éplucher la syntaxe de LINQ, mais de faire un tour de la technologie avec des exemples pratiques et simples. Nous n'allons ainsi pas continuer à taper du mapping à la main alors qu'il existe une façon bien plus simple et élégante d'utiliser LINQ avec une base de données, LINQ to Entities. C'est l'objet d'une section à venir, après quelques exemples de LINQ to Dataset et de LINQ to XML.

LINQ to Dataset

LINQ to Dataset est une extension de LINQ permettant de travailler directement sur le contenu de Datasets typés ou non. L'article est fourni avec un projet exemple qui montre quelques manipulations d'un Dataset typé, n'hésitez pas à « jouer » avec.

LINQ to Dataset est tout aussi puissant et versatile que les autres versions de LINQ. Je mettrai juste un bémol à l'utilisation de LINQ sur les Datasets : la bonne pratique (et la RAM de votre machine !) interdit de charger toute une base de données dans un Dataset... Dès lors ce dernier ne doit contenir qu'un sous-ensemble minimal des données de la base, ce qui implique de l'avoir chargé en effectuant déjà un premier filtrage. LINQ to Dataset se place donc après cette première sélection purement SQL pour ne travailler que sur les données en RAM. Cela, à mon sens, restreint bien entendu l'utilité de LINQ ici puisqu'un « bon » Dataset est un Dataset presque vide...

Mais qu'importe, puisqu'on peut se servir de LINQ aussi sur les Datasets, je fais confiance à votre imagination pour en tirer avantage !

Exemple

Reprenons la base de données Northwind et les tables **Customers** et **Orders** dont nous avons tiré un Dataset fortement typé par les mécanismes usuels de Visual Studio. La classe s'appelle **NorthwindDataset** et son schéma est décrit dans le fichier **xsd** accompagné du code source C# produit par VS.

Je passerais sur la façon de créer une instance de ce Dataset et de remplir les deux tables avec les données issues de la base, puisqu'il s'agit ici de procédés déjà connus depuis longtemps de ADO.NET.

Nous en arrivons ainsi à la requête LINQ to Dataset. Celle-ci va utiliser une jointure déclarative entre les clients et les commandes afin de sélectionner tous les clients (ainsi que leurs commandes) dont le nom du contact commence par la lettre A.

```
var query = from c in ds.Customers
            join o in ds.Orders on c.CustomerID equals o.CustomerID
            into orderlines
            where c.ContactName.StartsWith("A")
            select new
            {
                CustID = c.CustomerID,
                CustName = c.ContactName,
                Orders = orderlines
            };
```

Les entités retournées sont formées de l'identificateur du client, de son nom et de la liste de ses commandes. Ce type est anonyme, créé à la volée. On remarquera que nous avons choisi de donner des noms de propriétés différents des noms d'origines, par exemple l'identificateur sera appelé **CustID** et non plus **CustomerID**. De la même façon vous pouvez voir comment l'ensemble des commandes d'un client est « stocké » (virtuellement) dans une variable interne à la requête (**orderlines**) et comment cette grappe d'objets est passée dans l'objet résultat (propriété **Orders**). Tout cela n'a valeur que d'exemple de syntaxe et n'a aucun caractère obligatoire ou fonctionnel bien entendu.

La jointure entre les clients et les commandes est effectuée « à la main » en utilisant **join** d'une façon similaire à la syntaxe SQL de même fonction. Le résultat de cette jointure est nommé (**orderlines**) pour être facilement ré-exploité dans la requête. On notera que la jointure ne fonctionne que sur l'égalité et que pour marquer cette obligation LINQ force l'utilisation du mot **equals** au lieu d'une égalité de type **==**. Il s'agit ici d'éviter toute confusion. Si **==** était autorisé on pourrait être tenté d'utiliser un autre opérateur de comparaison or cela n'aurait pas de sens pour LINQ à cet endroit. En utilisant un mot clé particulier, LINQ marque ainsi une utilisation particulière et spécifique de l'égalité. Il ne s'agit pas d'une interprétation personnelle mais d'une explication donnée par un membre de l'équipe LINQ aux TechEd Microsoft à Barcelone en octobre dernier, c'est donc une vraie info, certes anecdotique, mais de première qualité☺.

LINQ to Dataset est une adaptation de LINQ travaillant en mémoire, de fait il n'y a pas de code SQL généré à visualiser. Je vous fais grâce de la sortie console qui liste tous les objets, cela ne vous dirait rien. Le projet exemple est de toute façon fourni avec l'article pour que vous puissiez expérimenter la chose par vous-mêmes.

Regardons maintenant comment, au lieu de créer la jointure en LINQ, nous pourrions exploiter la relation existante dans le Dataset typé :

```
var query2 = from c in ds.Customers
            where c.ContactName.StartsWith("A") &&
                  c.GetOrdersRows().Any(sv=> sv.ShipVia==2 &&
                                              sv.ShippedDate.Year>1997)
            orderby c.ContactName ascending
            select new { c.ContactName, Orders = c.GetOrdersRows() };
```

Ici la requête est plus complexe. Nous souhaitons obtenir une liste contenant des éléments (type anonyme) constitués d'un nom de contact client et des commandes de ce client. Cela est réglé par la partie **select** de l'instruction.

Toutefois nous souhaitons que seuls les clients dont le nom de contact commence par la lettre A soient sélectionnés. C'est la première partie du **where** qui s'en charge.

Mais pour compliquer les choses nous désirons que seuls les clients ayant au moins une commande passée après l'année 1997 et livrée via le mode 2 (peu importe ce que signifie ce code) soient sélectionnés. Cela est pris en charge par la seconde partie du **where** en utilisant la relation **GetOrdersRows** qui retourne les commandes du client en cours d'analyse par LINQ. Cette méthode a été générée automatiquement par Visual Studio dans le Dataset typé. On remarque l'utilisation du class helper **Any** provenant de LINQ et de ses paramètre sous la forme d'une expression Lambda.

Comme on le voit, LINQ to Dataset est tout aussi sophistiqué que les autres émanations de LINQ. En réalité d'ailleurs nous n'avons rien utilisé qui soit ici spécifique à LINQ to Dataset, les requêtes proposées en exemple auraient pu être effectuée avec LINQ to SQL notamment. Seule la source de données est différente (mapping d'une classe vers une table avec LINQ to SQL ou utilisation des tables d'un Dataset avec LINQ to Dataset). Et c'est bien là la force de LINQ, il en existe des versions spécialisées selon le type de la source de données, mais sa syntaxe et sa puissance ne changent pas.

LINQ to XML

Cette version de LINQ repose sur les mêmes bases que tout ce que nous avons vu jusqu'à maintenant mais elle introduit quelques éléments syntaxique propres. Sa spécificité est de pouvoir travailler sur des sources XML, voire de créer des fichiers XML. Plus de Dataset ni de **DataContext** ici, juste des documents XML, sur disque ou en mémoire.

Il est donc possible avec LINQ to XML d'interroger des documents XML de la même façon qu'un Dataset ou qu'une liste d'objet. Les exemples de syntaxe vus jusqu'ici sont applicables à des sources XML. Toutefois XML utilise un formalisme bien particulier, il existe donc toute une syntaxe LINQ to XML bien spécifique à ce contexte. Ce que nous allons voir maintenant.

L'un des gros avantages de LINQ to XML est de permettre l'interrogation et la création de documents XML en se passant totalement d'API plus ou moins complexes, et même de XPath, XQuery ou XSLT ! LINQ to XML n'a pas vocation à remplacer ces technologies, il se place juste au dessus pour donner au développeur un niveau d'abstraction supérieur et simplifier la manipulation des sources XML.

LINQ to XML propose un modèle particulier pour accéder aux données XML, les sources pouvant être un flux (*stream*), un fichier ou du XML en mémoire. En réalité il y a très peu de choses à savoir pour utiliser le modèle LINQ to XML, il suffit de connaître les types les plus courants de ce dernier qui sont : **XDocument**, **XElement** et **XAttribute**. Chacun possédant des constructeurs permettant de contrôler les objets créés avec précision. On comprend, bien entendu, rien que par leur nom la vocation de ces classes et leur relation hiérarchique calquant celle du formalisme XML (document / élément / attribut).

Créer des données XML

Regardons d'abord comment il est facile à l'aide des classes indiquées ci-dessus de créer un fichier XML par code :

```

public static void Demo1()
{
    XElement xml = new XElement("clients",
        new XElement("client",
            new XAttribute("ID", 2),
            new XElement("société", "e-naxos"),
            new XElement("site", "www.e-naxos.com")),
        new XElement("client",
            new XAttribute("ID", 5),
            new XElement("société", "Microsoft"),
            new XElement("site", "www.Microsoft.com")));
    Console.WriteLine(xml);
    Console.ReadLine();
}

```

La sortie console de cet exemple sera le contenu de la variable **xml** :

```

<clients>
  <client ID="2">
    <société>e-naxos</société>
    <site>www.e-naxos.com</site>
  </client>
  <client ID="5">
    <société>Microsoft</société>
    <site>www.Microsoft.com</site>
  </client>
</clients>

```

En utilisant **XElement** nous pouvons totalement créer le corps d'un fichier XML, mais pour satisfaire la norme XML ce corps peut être inclus dans un document (ce qui n'est pas indispensable pour le requêtage LINQ). Pour cela on utilise la classe **XDocument**. L'exemple suivant crée un fichier bien formé en exploitant cette possibilité, fichier qui est sauvegardé sur disque dans la foulée.

```

public static void Demo2()
{
    XDocument doc = new XDocument(
        new XDeclaration("1.0", "utf-8", "yes"),
        new XComment("Liste des clients"),
        new XElement("clients",
            new XElement("client",
                new XAttribute("ID", 2),
                new XElement("société", "e-naxos"),
                new XElement("site", "www.e-naxos.com")),
            new XElement("client",
                new XAttribute("ID", 5),
                new XElement("société", "Microsoft"),
                new XElement("site", "www.Microsoft.com"))));

    doc.Save("ClientsDemo.xml");

    Console.WriteLine(doc);
    Console.ReadLine();
}

```

Sur disque nous trouvons alors un fichier **ClientsDemo.xml** dont le contenu visionné par le bloc-notes est le suivant :

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--Liste des clients-->
<clients>
  <client ID="2">
    <société>e-naxos</société>
    <site>www.e-naxos.com</site>
  </client>
  <client ID="5">
    <société>Microsoft</société>
    <site>www.Microsoft.com</site>
  </client>
</clients>
```

LINQ to XML ne se limite ainsi pas uniquement à un requêtage simplifié des données XML, il permet aussi d'en créer très facilement.

Marier LINQ to XML avec LINQ to SQL/Dataset/Entities

Produire des données XML est aujourd'hui une activité courante pour une grande partie des applications. Produire ces données à partir d'autres données, notamment celles contenues dans une base SQL ou d'une liste d'objets en mémoire apparaît ainsi d'une grande utilité. Dès lors pourquoi ne pas marier LINQ to XML et sa possibilité de créer simplement des fichiers XML avec LINQ to Dataset, LINQ to SQL ou LINQ to Entities ?

Pourquoi pas en effet... Rien ne s'y oppose. Voyons dans un exemple comment réaliser en quelques lignes ce qui prendrait des pages de code avec d'autres méthodes.

Explications préalables : nous allons utiliser comme source la classe **Client** et le **DataContext** spécialisé de l'exemple de LINQ to SQL étudié plus haut dans cet article. Le code de ces déclarations ne sera pas repris ici pour alléger le texte.

```
public static void Demo3()
{
    const string northWind = "chaîne de connexion";
    using (DemoContext db = new DemoContext(northWind))
    {
        XElement xml = new XElement("clients",
            from c in db.Customers
            where c.ContactName.StartsWith("A")
            orderby c.ContactName
            select new XElement("client",
                new XAttribute("ID", c.CustomerID),
                new XAttribute("contact", c.ContactName))
        );
        Console.WriteLine(xml);
        xml.Save("ContactsDemo.xml");
    }
    Console.ReadLine();
}
```

Ce code crée un corps de document XML à partir des instances de la classe **Client** dont le nom de contact commence par la lettre A, la liste étant triée par ordre alphabétique de ces derniers. Le **select** crée les nouvelles entrées, la table elle-même est créée par le **XElement** de tête s'appelant **xml**. Pour créer un document bien formé et l'enregistrer sur disque par exemple, il suffit d'enchâsser le résultat **xml** dans un **XDocument** et de le

sauvegarder par la méthode vu précédemment. L'utilisation d'un **XDocument** n'est en rien indispensable et pour le prouver nous nous en passerons dans cet exemple.

La sortie console de l'exemple est la suivante :

```
<clients>
  <client ID="ROMEY" contact="Alejandra Camino" />
  <client ID="MORGK" contact="Alexander Feuer" />
  <client ID="ANATR" contact="Ana Trujillo" />
  <client ID="TRADH" contact="Anabela Domingues" />
  <client ID="GOURL" contact="André Fonseca" />
  <client ID="EASTC" contact="Ann Devon" />
  <client ID="LAMAI" contact="Annette Roulet" />
  <client ID="ANTON" contact="Antonio Moreno" />
  <client ID="FAMIA" contact="Aria Cruz" />
  <client ID="SPLIR" contact="Art Braunschweiger" />
</clients>
```

Magique non ?

Encore plus intéressant, surtout pour ceux qui, comme moi, restent réfractaires à la logique de Xpath ou XQuery que je n'utilise pas assez souvent pour être à l'aise les quelques fois où j'en aurais besoin, il y a bien entendu la possibilité d'interroger un fichier XML et même de transformer des données (un peu comme XSLT). Par exemple analyser une grappe XML, ne prendre que les éléments qui nous intéressent et créer une liste d'instances de la classe **Client** !

Dans le code qui suit nous prendrons le fichier **ContactsDemo.xml** créé à l'étape précédente et nous allons l'interroger pour créer une liste de string :

```
public static void Demo4()
{
    XDocument xml = XDocument.Load("ContactsDemo.xml");
    var query = from c in xml.Descendants("client")
                where ((string)c.Attribute("ID")).Contains('M')
                orderby (string)c.Attribute("contact") descending
                select (string)c.Attribute("contact")+" ["+(string)c.Attribute("ID")+"];

    Console.WriteLine("il y a "+query.Count().ToString()+" clients:\n");
    foreach (string s in query) Console.WriteLine(s);
    Console.ReadLine();
}
```

On remarque d'abord qu'en l'absence d'un schéma la requête LINQ to XML fonctionne en utilisant directement les noms des attributs. Sans schéma pas d'aide sur ces derniers qu'il faut taper comme des chaînes et transtyper convenablement.

La sortie console est la suivante :

Le résultat compte 4 clients:

```
Aria Cruz [FAMIA]
Annette Roulet [LAMAI]
Alexander Feuer [MORGK]
Alejandra Camino [ROMEY]
```

Bluffant non ? LINQ to XML C'est un petit pas pour XML mais un pas de géant pour le développeur !

Les grincheux et les impatientes diront peut-être qu'on s'éloigne du typage fort qu'offre LINQ puisqu'ici il faut manipuler des balises dont les noms sont saisis en chaînes de caractères (risque d'erreur) et dont les types doivent être obtenus par transtypage (perte du typage fort).

Certes... Mais j'ai une bonne nouvelle pour eux (et pour les autres !), cela s'appelle LINQ to XSD, tout simplement. Il y a déjà eu plusieurs releases en preview de cette technologie qui avait été initiée par le docteur Ralf Lämmel quand il appartenait à l'équipe LINQ to XML. Ce chercheur étant reparti dans sa Germanie natale pour y professer, ce projet a connu quelque retard. Mais très récemment Shyam Pather (Microsoft) à la conférence XML 2007 de Boston a annoncé que tous les efforts seraient faits pour fournir LINQ to XSD, mais dans un second temps, après la sortie de Visual Studio 2008. Encore un peu de patience donc et il sera possible de déférer les propriétés en bénéficiant du support Intellisense avec LINQ to XSD.

LINQ to Entities

Jusqu'à ce point nous avons pu voir à l'œuvre LINQ sur des collections, sur des données XML, des Datasets et des sources SQL. Ce sont ces dernières qui nous intéressent à nouveau ici.

LINQ to SQL est très puissant et simple à utiliser, mais il s'agit d'une approche centrée sur les données. En réalité LINQ to SQL apparaît dès lors comme une simplification de l'écriture des Data Access Layer en rendant le mapping plus aisé.

Avec LINQ to Entities les données sources ne sont plus des tables ou des objets mappés sur ces dernières, les sources sont purement conceptuelles. Les équipes de LINQ ont réussi à casser le mur entre langage objet et structure de base de données.

Un constat navrant s'impose : une base de données apparaît toujours au développeur sous la forme de son schéma physique alors qu'une base bien conçue a d'abord été murement réfléchi par un développeur spécialisé ou un analyste. Et ce que ces derniers ont modélisé (le mot est lâché) c'est un MCD, un Modèle Conceptuel des Données. Ce MCD a pu être conçu dans un style Merise, ou en notation Entité/Relation, sous la forme de diagrammes UML de classes retranscrits en schémas physiques, peu importe l'outil et la méthodologie utilisée, *une base de données naît d'abord sous les traits d'un modèle conceptuel et non d'un schéma physique.*

La plupart des outils de conception de schémas de base de données savent d'ailleurs produire automatiquement le schéma physique (MPD : Modèle Physique des Données) à partir du MCD, jusqu'au script SQL de création de la base qui n'est qu'un sous-produit de toute cette démarche, le côté « sale », prosaïque, terriblement matériel qui n'intéresse guère de monde à tel point qu'on délègue ce travail à des automates...

Et pourtant ! C'est avec ce schéma physique, cet *avorton* du MPD produit par un automate que le développeur d'applications va devoir se frotter en permanence ! Le rôle du développeur revient ainsi à « défaire » le travail de l'automate pour « remonter » sur le MCD afin de le faire coïncider avec les concepts, objectifs, de son application. Une tâche

stupide, semée d'embûches et source d'innombrables erreurs (même si le développeur peut disposer du MCD original de la base de données, cela ne simplifie en rien sa tâche de programmation des accès aux données).

Développeur != Plombier

Les développeurs ne sont pas des plombiers (quelle que soit par ailleurs la noblesse de ce métier) et pourtant ils ont la nette sensation qu'en matière de base de données ils passent leur temps à abouter, tordre et remodeler de la tuyauterie dans l'espoir de la faire coïncider aux besoins de leurs applications. Une gymnastique usante et source d'erreurs entre un espace rectangulaire, celui des SGBD, et la logique des graphes et des objets propre aux langages modernes.

Cette sensation d'être tout le temps en train de « bricoler » est, au-delà du constat technique navrant, quelque chose de peu gratifiant. Et quand un développeur ne se sent pas gratifié par ce qu'il fait il le fait mal.

Changer de dimension

Il faut donc trouver un moyen d'échapper à la logique en 2D des rectangles du docteur Codd.

Lorsqu'on regarde un diagramme de classes d'une application objet bien conçue, on s'aperçoit qu'il n'y a que fort peu de différences avec un MCD de base de données. Au départ ces deux mondes sont finalement très proches ! Le divorce n'est qu'une apparence, un terrible coup du sort : le concepteur de la base est obligé « d'abaisser » le niveau conceptuel de son schéma à celui d'un schéma physique pour satisfaire les exigences du SGBD cible. Le développeur d'applications ne voyant que ce résultat...

Il faut ainsi trouver un moyen pour que la base de données puisse réapparaître sous une forme conceptuelle afin que le développeur ne s'encombre plus des détails et limitations du modèle physique optimisé pour le moteur de base de données et non pour un humain ou une application.

On pourrait supposer créer un outil qui, partant d'un diagramme de classes, produirait d'un côté le schéma physique de la base de données et, de l'autre, le mapping vers les objets en mémoire. En fait cela existe déjà mais n'offre finalement pas la souplesse attendue.

En effet, une base de données répond à des critères normatifs issus des règles de Codd, et rien ne permet d'y échapper. Un bon designer de base de données, même lorsqu'il utilise un outil de conceptualisation, aura toujours tendance à créer des entités et des relations qui bien que modélisant parfaitement les besoins de l'utilisateur resteront empreintes d'une certaine « logique rectangulaire ». Par exemple l'héritage n'existe pas dans les bases SQL, même si certains outils permettent cette notation. Ne parlons pas du support des interfaces, des class helpers, etc... De plus une base de données ne contient que... des données ! Il manque toute la logique des actions (méthodes), l'ordre des séquences, bref tout ce qui fait qu'une application est autrement plus complexe qu'une base de données.

Cette dynamique particulière entre données et actions est le propre de la programmation objet, certainement pas de la création des bases de données ni même de leur modélisation et encore moins de leur optimisation (qui impose souvent de dégrader encore plus le

schéma conceptuel et le modèle physique par des processus barbares nommés « dénormalisation »).

De fait, obliger le partage d'un même schéma conceptuel pour la base de données et l'application est une fausse bonne idée. Retour à la case départ.

La solution consiste finalement à laisser les designers de base de données exercer au mieux leur art, et à laisser les développeurs exercer le leur. C'est le principe de LINQ to Entities : à partir du schéma physique de la base de données le développeur va pouvoir créer un modèle conceptuel collant parfaitement aux besoins de son application. Mieux, il pourra créer autant de modèles différents sur un même schéma physique en fonction des concepts qui sont manipulés par telle ou telle autre applications.

LINQ to Entities est ainsi une version de LINQ qui permet d'écrire des requêtes non plus en direction d'une base de données et de son schéma physique, mais bien en direction d'un modèle totalement conceptuel qui va, en quelque sorte, faire écran entre la base de données et les objets de l'application. Charge à LINQ de traduire en SQL tout ce que le développeur fera avec les instances des classes décrites dans le modèle.

Passons à la réalité...

Il était impossible de parler de LINQ sans quelques digressions sur les concepts en jeu tellement ils sont essentiels et lourds de conséquences. Mais dans cet article le moment est venu de passer à un exemple concret !

Avertissement : Ce que nous allons voir maintenant est en réalité une extension de Linq to SQL. Dans ce mode on dispose d'un modèle, d'un concepteur visuel spécifique et d'une génération automatiquement de code. Mais en réalité on reste encore en Linq to SQL. Le véritable Linq to Entities repose sur l'Entity Framework. Les nuances visuelles pour la démo sont faibles, surtout pour un « tour d'horizon », ce qu'est le présent article. Le véritable Linq to Entities est encore en bêta (la 3 est disponible sur MSDN à l'heure où j'écris ce texte), les différences semblent mineures mais sont essentielles techniquement. Linq to Entities repose sur un mapping XML et non sur des attributs, il permet aussi qu'une entité du modèle soit mappée sur plusieurs tables ou vues ce qui est impossible avec Linq to SQL. La démo qui suit utilise ainsi le mode visuel de Linq to SQL pour faire comprendre ce qu'est Linq to Entities, en raison de cette proximité visuelle et pratique et aussi parce que ce mode de fonctionnement de Linq to SQL est déjà en soi une petite révolution. Mais techniquement, vous l'aurez compris, Linq to Entities et l'Entity Framework vont encore plus loin, même si pour l'instant ce n'est qu'une bêta... (présentation sur MSDN : <http://msdn2.microsoft.com/en-us/library/bb896679.aspx>).

Tout d'abord, LINQ to Entities ne réclame au départ rien de spécial, on peut donc l'utiliser dans tout type de projet. Pour créer le modèle conceptuel, appelé EDM (Entity Data Model, modèle de données des entités), il suffit, au sein d'un projet existant, d'ajouter un nouvel élément et de choisir dans le menu proposé « ADO.NET Entity Model ». L'Entity Framework étant encore en bêta test (voir l'avertissement plus haut) nous utiliserons « LINQ to SQL classes » (voir la figure ci-dessous). Il y a très peu de différences visuelles et nous reviendrons dans un autre article sur les « véritables » EDM de l'Entity Framework. Considérez ici que le principe est le même, la principale nuance étant que Linq to Entities et l'Entity Framework forment une solution totalement objet autorisant un

mapping XML très libre et beaucoup moins limité que celui des classes Linq to SQL de la démo qui suit. Je reviendrai en détail sur l'Entity Framework et Linq to Entities dans un article à venir.

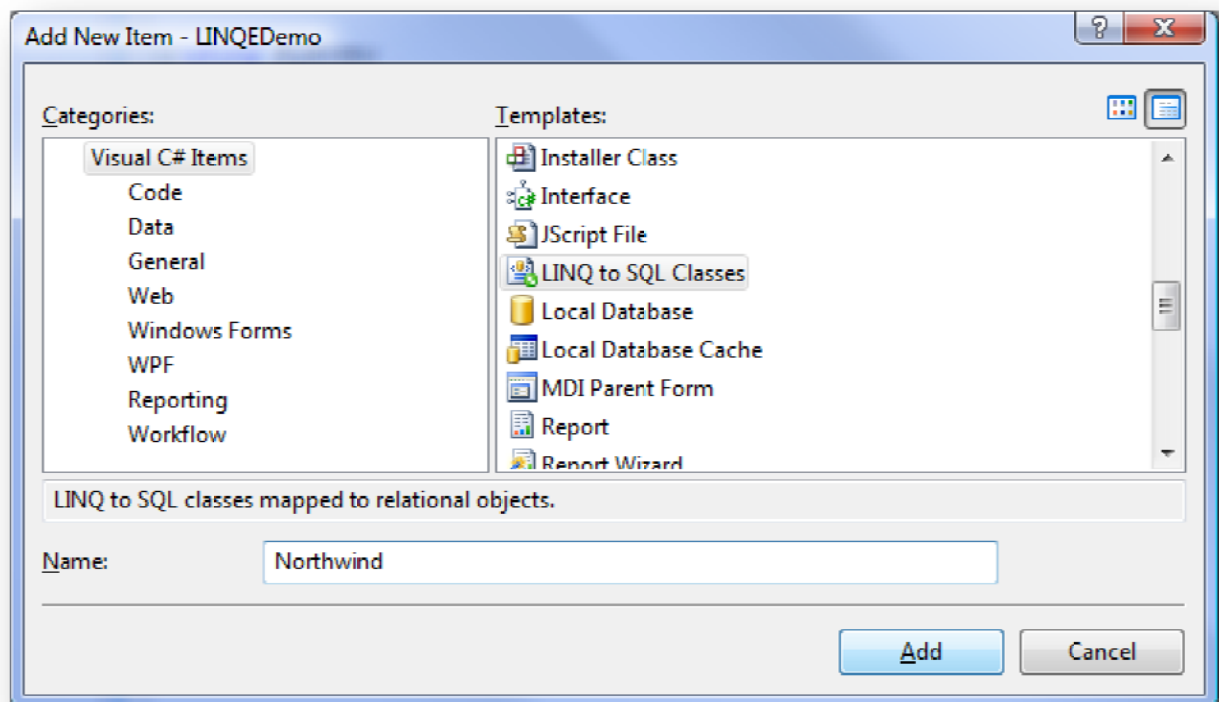


Figure 1 - Création d'un diagramme de classes Linq to SQL

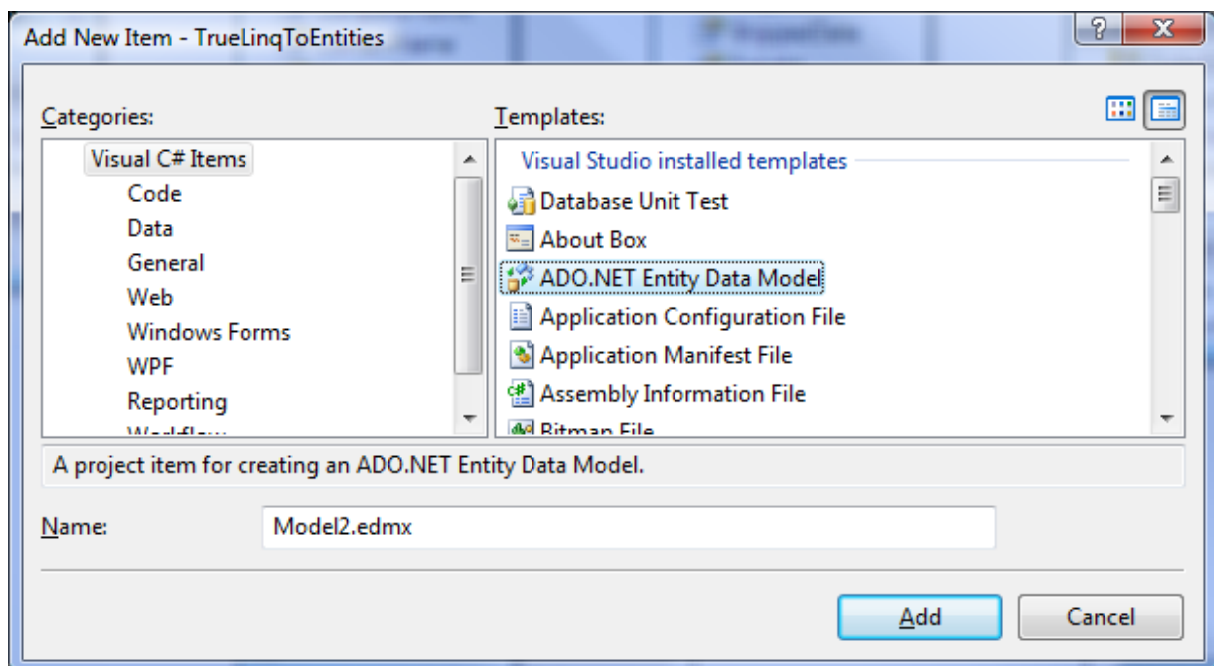


Figure 2 - Création d'un diagramme de classes ADO.NET Entity Data Model

Construire le modèle

La construction d'un modèle s'effectue de façon totalement libre, comme un diagramme de classes UML et avec des outils très proches. Il est donc possible, partant de rien, de concevoir un modèle correspondant exactement au besoin de l'application et dans un second temps de mapper les classes ou propriétés à des tables et champs d'une base de données. Les classes Linq to SQL utilisées pour cette démo n'autorisent qu'un mapping 1:1 (1 classe = 1 table / vue). Linq to Entities et l'Entity Framework permettent eux le mapping multiple démultipliant les possibilités et *renforçant le découplage avec le modèle physique de la base de données*.

Il existe aussi une façon plus simple de procéder que je vais utiliser ici, il suffit de partir d'une connexion avec une base de données et de piocher par drag'n drop les tables qu'on désire adresser puis de modifier le modèle pour le faire tendre vers le niveau d'abstraction de l'application (les modèles ADO.NET Entity models disposent d'un wizard permettant de construire un diagramme depuis une base de données automatiquement) .

En partant de la base Northwind et en sélectionnant les tables Customers et Orders nous arrivons à une première ébauche de modèle. On notera que le designer visuel des classes Linq to SQL dispose d'une toolbox permettant d'ajouter des classes, des associations et des héritages. On retrouve les mêmes fonctions dans les modèles Entity Framework, sauf que le concept de classe est remplacé par celui d'entité.

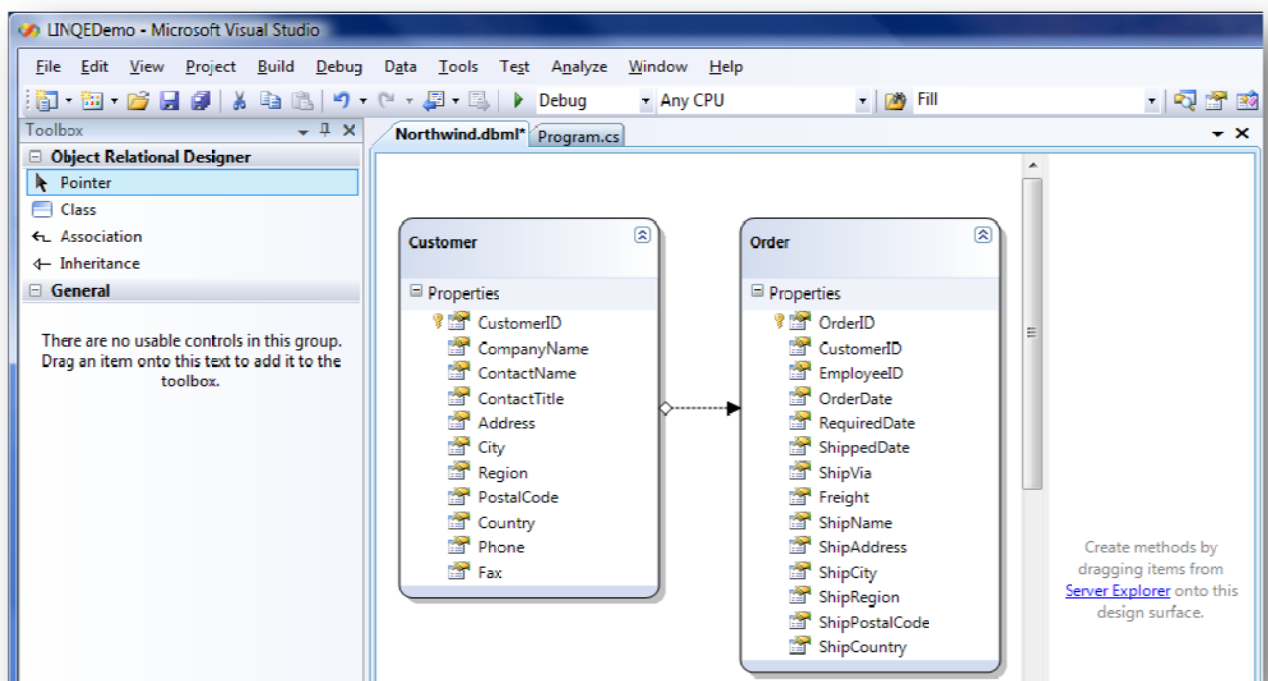


Figure 3 - Un modèle de classes Linq to SQL

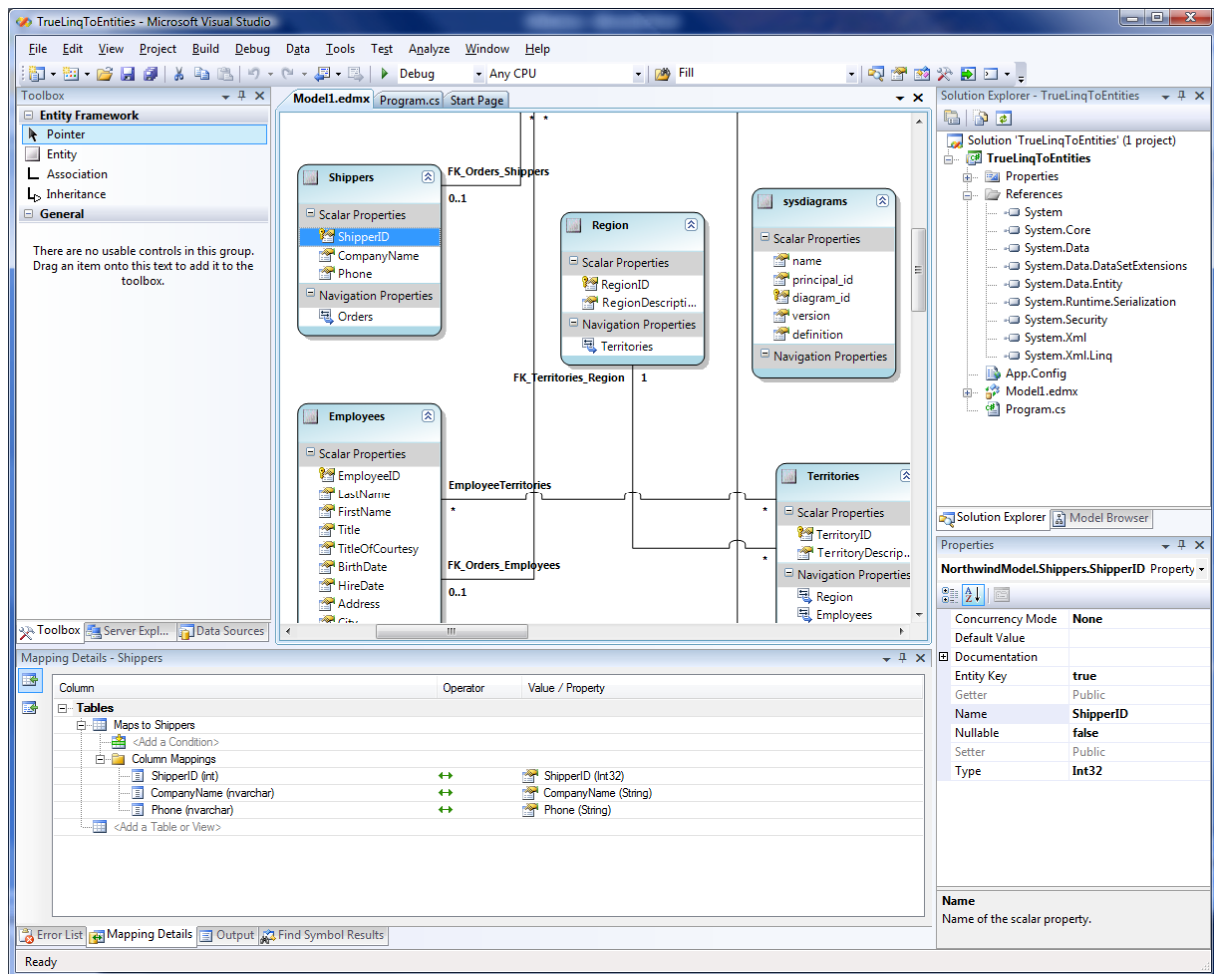


Figure 4 - Un diagramme EDM, Entity Data Model

Le diagramme présenté figure 4 montre le concepteur visuel des EDM. On remarque que si les outils et la présentation sont similaires, on dispose, sous le diagramme, d'un onglet spécifique permettant de gérer le mapping. C'est cet outil là plus qu'un autre qui, d'un point de vue pratique, change tout puisqu'il permet de manipuler le mapping librement là où les modèles de classes Linq to SQL ne le permettent pas. Au passage vous remarquerez sur le schéma de la figure 4 des relations many-to-many qui ne sont pas prises en charges par les classes Linq to SQL.

A ce stade précis notre modèle de la figure 3 est « brut de fonderie » il calque à 100% le modèle physique puisque nous venons justement de partir de ce dernier. On remarque que par exemple la jointure entre clients et commandes a été automatiquement déduite de celle existante dans la base de données. On remarque aussi que le designer visuel a créé deux classes, enlevant automatiquement au passage le « s » final de Customers et Orders...

Les propriétés de la jointure sont les suivantes :

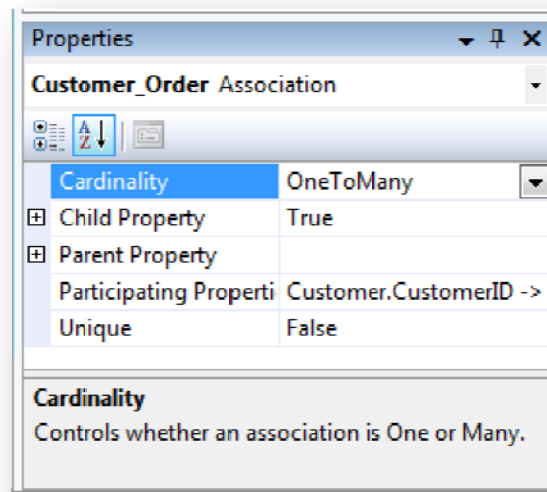


Figure 5 - Propriété d'une relation (Linq to SQL)

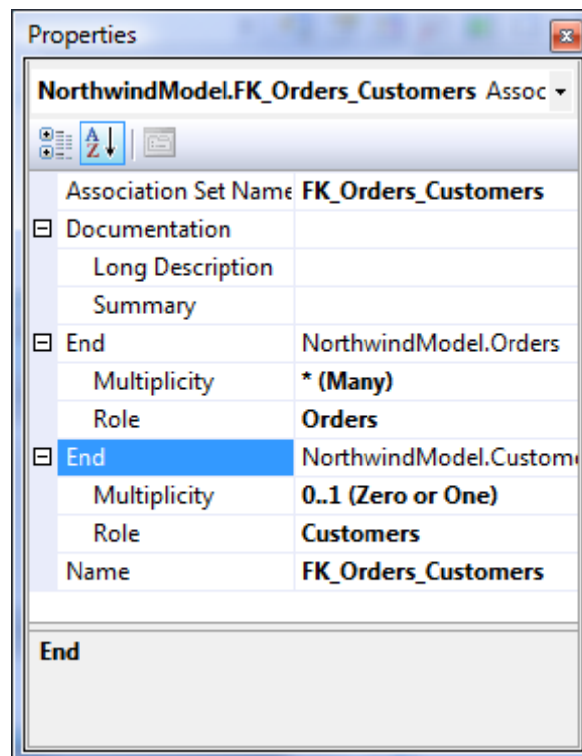


Figure 6 - La même relation exprimée dans un Entity Data Model

Il s'agit ici de définir une relation entre deux classes, comme dans un diagramme de classes UML et non plus une jointure SQL. Notre exemple s'appuie sur Linq to SQL comme indiqué dans l'avertissement d'introduction mais nous continuons à présenter, quand cela est possible, un parallèle avec l'Entity Framework. On remarque ainsi les petites nuances dans la définition d'une relation entre la figure 5, modèle de classes Linq to SQL, et la figure 6, modèle de type Entity Data Model.

Personnaliser le modèle

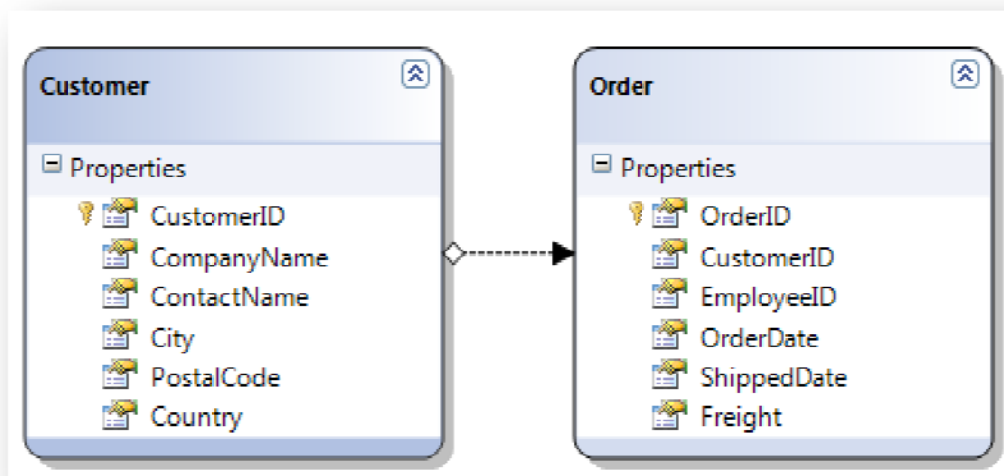
De façon très simple il est possible d'ajouter ou de supprimer des relations, des tables, d'enlever ou d'ajouter des propriétés aux entités, etc...

Il est même possible de modifier le mapping des champs, d'inspecter et de changer le SQL qui sera généré par LINQ et même d'indiquer à ce dernier qu'il faut non pas produire du SQL pour accéder aux données mais utiliser une ou plusieurs procédures stockées (sélection et mise à jour notamment). Les possibilités les plus avancées ne sont disponibles que dans un modèle de type Entity Data Model. Les modèles de classes Linq to SQL restent plus limités en terme de mapping notamment.

LINQ to Entities est une technologie simple mais tellement riche qu'il est bien évidemment hors du cadre du présent article d'en faire le tour complet et en détail. D'autant que cette technologie ainsi que l'Entity Framework ne sont pour l'instant qu'en bêta. Il est plus simple de continuer notre démonstration en se basant sur les modèles de classes Linq to SQL, technologie intégrée à Visual Studio 2008 et utilisable tout de suite en production.

Pour illustrer le propos sans entrer dans la complexité d'une véritable application, apportons quelques modifications « symboliques » à notre modèle de classes Linq to SQL en supprimant des propriétés et en complétant les entités du modèle par du code personnalisé (on pourrait aller encore plus loin dans un modèle EDM).

En premier lieu nous ne conservons que quelques informations pour chaque entité, ce qui produit le diagramme suivant :



Dans un second temps nous allons ajouter du code à chaque entité. Un clic droit et « voir le code » affiche l'éditeur de code placé sur la définition de l'entité. On remarque que tout le code automatique est caché dans un autre fichier et que les classes présentées sont vides (classes partielles). Pour faire simple modifions les méthodes **ToString()** de chaque entité :

```

namespace LINQEDemo
{
    partial class Customer
    {
        public override string ToString()
        {
            return "Société " + CompanyName;
        }
    }

    partial class Order
    {
        public override string ToString()
        {
            return "Cde n° " + OrderID + " Pour: " + Customer.CompanyName;
        }
    }
}

```

On imagine bien qu'il est possible d'ajouter tout le code fonctionnel dont on a besoin, on est en C# dans du code « normal » et bénéficiant de l'ensemble des possibilités du Framework. Nous nous arrêterons ici pour les personnalisations.

Premier test du modèle

Pour tester notre modèle et ses personnalisations, le plus simple est de demander une fiche client et une fiche commande, isolément, et d'en faire un affichage en chaînes de caractères (ce qui appellera les méthodes **ToString** surchargées) :

```

public static void Demo1()
{
    using (NorthwindDataContext db = new NorthwindDataContext())
    {
        Console.WriteLine(db.Orders.First());
        Console.WriteLine(db.Customers.First());
        Console.ReadLine();
    }
}

```

La sortie console nous montre alors :

```

Cde n° 10248 Pour: Vins et alcools Chevalier
Société Alfreds Futterkiste

```

La première ligne affiche la première commande de la base de données mise en forme par notre surcharge de **ToString()**. Il en va de même pour le premier client.

Au passage on remarque que la technique est similaire à LINQ to SQL puisque nous utilisons un **DataContext** personnalisé. La seule différence est que ce dernier est généré automatiquement par le designer de Visual Studio. C'est bien parce que nous sommes toujours sous LINQ to SQL... la véritable révolution se trouve dans l'Entity Framework et Linq to Entities que nous avons survolé ici. Sous Entity Framework, un EDM donne naissance à un **ObjectContext** et non plus un **DataContext**. La différence semble mineure, mais elle est techniquement profonde et ouvre des possibilités bien plus grandes que celles de Linq to SQL.

Ajout d'une procédure stockée au modèle

Par drag'n drop il est possible de déposer sur un modèle de classes Linq to SQL des procédures stockées depuis le gestionnaire de serveurs vers un espace vertical collé au modèle. Cela ajoute la PS à ce dernier en tant que méthode du **DataContext** personnalisé... Ajoutons ainsi la PS **Ten_Most_Expensive_Products** et interrogeons-là pour tester le fonctionnement de l'ensemble :

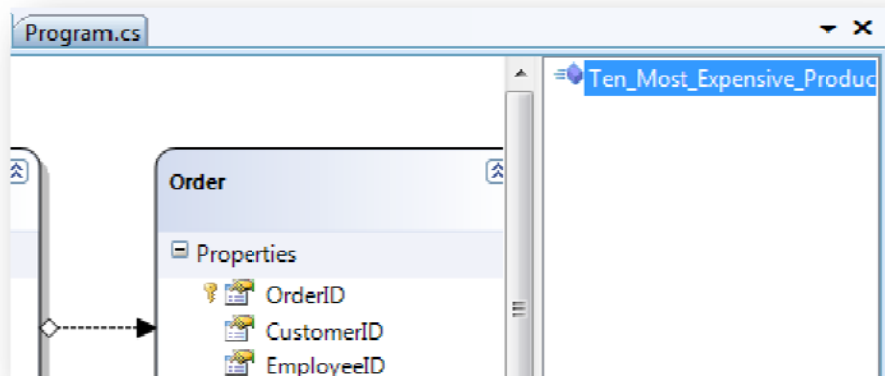


Figure 7 - L'ajout d'une procédure stockée à un modèle

On voit sur l'image ci-dessus la procédure stockée ajoutée dans la colonne de droite, collée au modèle (les procédures stockées sont gérées un peu différemment sous Entity Framework).

Le code pour activer cette méthode est on ne peut plus simple (la classe **ObjectDumper** est la même que celle utilisée plus avant dans cet article) :

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
    ObjectDumper.Write(db.Ten_Most_Expensive_Products());
    Console.ReadLine();
}
```

Code qui produit la sortie console suivante :

```
TenMostExpensiveProducts=Côte de Blaye    UnitPrice=263,5000
TenMostExpensiveProducts=Thüringer Rostbratwurst    UnitPrice=123,7900
TenMostExpensiveProducts=Mishi Kobe Niku    UnitPrice=97,0000
TenMostExpensiveProducts=Sir Rodney's Marmalade    UnitPrice=81,0000
TenMostExpensiveProducts=Carnarvon Tigers    UnitPrice=62,5000
TenMostExpensiveProducts=Raclette Courdavault    UnitPrice=55,0000
TenMostExpensiveProducts=Manjimup Dried Apples    UnitPrice=53,0000
TenMostExpensiveProducts=Tarte au sucre    UnitPrice=49,3000
TenMostExpensiveProducts=Ipoh Coffee    UnitPrice=46,0000
TenMostExpensiveProducts=Rössle Sauerkraut    UnitPrice=45,6000
```

On peut de la même façon ajouter et gérer des PS qui attendent des paramètres. C'est le cas dans la base de données Northwind de la procédure **CustOrderHist** qui, une fois ajoutée au modèle, donnera naissance à une méthode de même nom attendant en paramètre un ID client.


```

using (NorthwindDataContext db = new NorthwindDataContext())
{
    db.Log = Console.Out;
    ObjectDumper.Write(db.CustOrderHist("ALFKI"));
    Console.ReadLine();
}

```

Nous avons ajouté ici la sortie de la trace SQL sur la console. De fait, pour le client indiqué la sortie console est la suivante (avec en premier le SQL produit par LINQ) :

```

EXEC @RETURN_VALUE = [dbo].[CustOrderHist] @CustomerID = @p0
-- @p0: Input NChar (Size = 5; Prec = 0; Scale = 0) [ALFKI]
-- @RETURN_VALUE: Output Int (Size = 0; Prec = 0; Scale = 0) [Null]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

ProductName=Aniseed Syrup          Total=6
ProductName=Chartreuse verte      Total=21
ProductName=Escargots de Bourgogne Total=40
ProductName=Flotemysost           Total=20
ProductName=Grandma's Boysenberry Spread Total=16
ProductName=Lakkalikööri        Total=15
ProductName=Original Frankfurter grüne Soße Total=2
ProductName=Raclette Courdavault  Total=15
ProductName=Rössle Sauerkraut     Total=17
ProductName=Spegesild             Total=2
ProductName=Veggie-spread         Total=20

```

Nous disposons ainsi de l'historique des commandes d'un client sous la forme d'une simple méthode du DataContext et notre application peut utiliser directement le résultat sous la forme d'une collection typée.

Utiliser les relations

Il est temps d'interroger le modèle en exploitant les relations qu'il définit. Voici une requête un petit plus complexe :

```

public static void Demo4()
{
    using (NorthwindDataContext db = new NorthwindDataContext())
    {
        db.Log = Console.Out;
        var query = (from c in db.Customers
                    select new
                    {
                        c.CompanyName,
                        c.City,
                        Orders = from o in c.Orders
                                where o.ShipCode == 2
                                select new { o.OrderID, o.ShipCode }
                    }).Take(5);

        ObjectDumper.Write(query, 2);
        Console.ReadLine();
    }
}

```

Explications : nous demandons ici à LINQ de nous fournir une liste constituée d'éléments dont le type, anonyme, est formé de trois propriétés : le nom de la société du client, la

ville du client et une propriété **Orders** qui est elle-même une liste. Cette liste est constituée elle-aussi d'éléments de type anonyme possédant deux propriétés, l'ID de la commande et le mode de livraison. La propriété **Orders** est fabriquée sur la base d'une sous-requête LINQ filtrant les commandes de chaque client et ne retenant que celles ayant un mode de livraison égal au code 2. Enfin, nous ne prenons que les 5 premiers clients de la liste globale (par l'opération **Take(5)** appliquée à toute la requête mise entre parenthèses).

Je vous fais grâce de la sortie console qui montre outre le résultat attendu les diverses requêtes SQL envoyées à la base de données par LINQ (grâce à **db.Log** redirigée vers la console).

Créer des relations

Si nous ajoutons la table **Suppliers** (fournisseurs) à notre modèle nous disposons d'une entité de plus dans ce dernier mais aucune relation n'est déduite de la base de données. En effet, il n'existe aucune jointure entre commandes et fournisseurs ni entre ces derniers les clients. Il existe bien entendu dans la base de données un chemin reliant toutes ces entités en passant par les lignes de commandes et les articles. Mais d'autres chemins sont envisageables.

De plus, nous travaillons sur un modèle conceptuel propre à notre application, nous ne sommes que très peu intéressés par la réalité de l'organisation de la base de données. Notamment, nous souhaitons ici obtenir une liste contenant pour chaque client la liste des fournisseurs se trouvant dans la même ville. Cela n'a pas de sens pour la base de données telle qu'elle a été conçue, mais cela en a un pour nous, ponctuellement dans notre application (par exemple étudier comment faire livrer les clients directement par les fournisseurs les plus proches au lieu de faire transiter la marchandise par notre stock...).

Nous allons pour cela créer une relation « à la volée » dans une requête :

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
    var query = from c in db.Customers
                join s in db.Suppliers on
                c.City equals s.City into sups
                where sups.Count() > 0
                select new { c.CompanyName, c.City, Suppliers = sups };
    ObjectDumper.Write(query, 2);
    Console.ReadLine();
}
```

La sortie console ne vous ferait rien voir de plus, mais regardons le code SQL généré et méditez sur la différence de lisibilité entre les quelques lignes LINQ ci-dessus et ce pavé SQL ci :

```

SELECT [t0].[CompanyName], [t0].[City], [t1].[SupplierID], [t1].[CompanyName] AS
[CompanyName2], [t1].[ContactName], [t1].[ContactTitle], [t1].[Address], [t1].[
City] AS [City2], [t1].[Region], [t1].[PostalCode], [t1].[Country], [t1].[Phone]
, [t1].[Fax], [t1].[HomePage], (
    SELECT COUNT(*)
    FROM [dbo].[Suppliers] AS [t3]
    WHERE [t0].[City] = [t3].[City]
) AS [value]
FROM [dbo].[Customers] AS [t0]
LEFT OUTER JOIN [dbo].[Suppliers] AS [t1] ON [t0].[City] = [t1].[City]
WHERE ((
    SELECT COUNT(*)
    FROM [dbo].[Suppliers] AS [t2]
    WHERE [t0].[City] = [t2].[City]
)) > @p0
ORDER BY [t0].[CustomerID], [t1].[SupplierID]
-- @p0: Input Int (Size = 0; Prec = 0; Scale = 0) [0]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

```

Je vous laisse faire vos propres déductions...

Modifications des données

Interroger de façon simple et contrôlée les données est une chose, que LINQ to Entities fait fort bien, mais qu'en est-il de la modification des données ?

Si vous vous attendez à des tonnes de code vous allez être déçus !

Modifions une fiche client pour commencer :

```

public static void Demo6()
{
    using (NorthwindDataContext db = new NorthwindDataContext())
    {
        var cust = db.Customers.First(c => c.CustomerID == "PARIS");
        cust.PostalCode = "75016";
        db.SubmitChanges();
    }
}

```

Nous obtenons le client dont l'ID est « PARIS », nous modifions son code postal puis nous sauvegardons les modifications dans la base de données... Plus simple je n'ai pas en stock. En revanche je peux vous montrer le code SQL généré par LINQ pour ces trois petites lignes de C# :

```

SELECT TOP (1) [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[City], [t0].[PostalCode], [t0].[Country]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[CustomerID] = @p0
-- @p0: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [PARIS]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

UPDATE [dbo].[Customers]
SET [PostalCode] = @p6
WHERE ([CustomerID] = @p0) AND ([CompanyName] = @p1) AND ([ContactName] = @p2)
AND ([City] = @p3) AND ([PostalCode] = @p4) AND ([Country] = @p5)
-- @p0: Input NChar (Size = 5; Prec = 0; Scale = 0) [PARIS]
-- @p1: Input NVarChar (Size = 17; Prec = 0; Scale = 0) [Paris spécialités]
-- @p2: Input NVarChar (Size = 14; Prec = 0; Scale = 0) [Marie Bertrand]

```

```
-- @p3: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [Paris]
-- @p4: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [75012]
-- @p5: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [France]
-- @p6: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [75016]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

Personnellement j'ai une petite préférence pour la version LINQ. Pas vous ?

Il est bien entendu possible de continuer comme ça très longtemps avant d'avoir fait le tour de LINQ. Par exemple ajouter une commande à un client (il suffit de créer un objet **Order**, de l'initialiser, de l'ajouter par **Customer.Add()** à l'objet client et de sauvegarder les changements...).

LINQ to Entities propose des mécanismes autrement sophistiqués notamment dans le tracking des modifications faites aux objets et à l'identification de ces derniers. Car bien entendu les grappes ou les objets isolés que nous récupérons n'apparaissent que comme de simples instances pour le développeur, mais au moment de mettre à jour les données comment LINQ sait-il que ces objets proviennent bien de la base de données, que le client 1256 dont on a changé l'ID (cela arrive) est bien le 1256 dans la base (alors que nous n'avons pas nous-mêmes conservé l'ancienne valeur) ? etc, etc...

D'autres questions du même type se posent lorsqu'il s'agit d'envoyer un objet en dehors de la machine (par du remoting par exemple) puis de le récupérer modifié et d'appliquer les changements à la base de données. On pourra aussi se demander comment les transactions sont gérées par LINQ.

Tout cela a bien entendu des réponses, mais nous dépasserions largement les limites de cet article de présentation dont les proportions ont explosé au fil de ce voyage que je voyais un petit plus court au départ...

N'hésitez pas à consulter mon blog, certains billets répondent aux questions soulevées ici et d'autres réponses viendront s'ajouter avec le temps comme, bien entendu, un article totalement consacré à Linq to Entities et l'Entity Framework.

Conclusion

J'ai bien cru ne jamais atteindre ce mot magique ! Conclusion.

Il est effectivement temps de conclure. LINQ est une déesse polymorphe dont on ne saurait se lasser de décrire les courbes gracieuses, mais il n'est de bonne compagnie qui ne se quitte, alors je vous dis à bientôt pour d'autres articles qui visiteront plus en détail d'autres aspects de Visual Studio 2008 et du Framework 3.5...

Comme je le conclus souvent sur mon blog : Stay tuned !