



Formations .NET C# – Delphi.NET / Win32
Audit – Développement

© Copyright 2006 Olivier DAHAN
Reproduction, utilisation et diffusion interdites sans
l'autorisation de l'auteur. Pour plus d'information
contacter odahan@e-naxos.com

ADO.NET « real life^{*} »

Accès concurrentiels, mises à jour de données hiérarchisées, verrous,
transactions, rapport IHM/données, ...

^{*} « dans la vie réelle »

Abstract

ADO.NET est un middleware d'accès aux données aujourd'hui mature offrant tout ce qui est nécessaire et utile à la programmation d'applications fonctionnant aussi bien sur Windows que sur des smart-phones ou via le Web.

Toutefois derrière la liste de ses fonctionnalités et au-delà de sa versatilité se cache des monstres coriaces : le mode déconnecté et la gestion des accès concurrents. Véritables casse-têtes que ADO.NET n'aide pas forcément à résoudre, voire qu'il complique à souhait en forçant l'emploi d'une logique de verrous optimistes qui ne peuvent satisfaire les utilisateurs finaux tant les frustrations en découlant sont nombreuses.

Il apparaît donc comme nécessaire de développer des stratégies d'utilisation de ADO.NET permettant à la fois de bénéficier de ces immenses services et de satisfaire les utilisateurs de nos applications.

ADO.NET « real live », c'est justement le sujet de cet article.

Après une partie plus théorique nous aborderons la réalisation d'une application en nous plaçant non plus sous l'angle de l'illustration d'un propos technique mais dans le contexte d'une application réelle, ce qui change tout comme vous le constaterez.

Note de l'auteur :

Le présent article s'est étoffé au fur et à mesure de son écriture pour atteindre une taille plus que respectable. Découlant de cette situation, l'écriture s'est étalée sur plusieurs jours souvent espacés les uns des autres. Ces conditions n'ont pas permis d'assurer à l'édifice un niveau de cohérence à la hauteur des exigences de l'auteur. Sachez qu'il en a conscience et qu'il s'en excuse. Il espère que l'utilité des informations ici publiées saura vous faire oublier les imperfections du document.

Sommaire

Présentation	1
Une histoire de verrous.....	1
Les verrous du mode pessimiste	1
Le mode optimiste sans verrou	2
C'est le dernier qui a parlé à raison.....	3
Mieux vaut prévenir que guérir !	5
Les conflits sans transactions	5
Les relations maître / détail	6
La base exemple	7
Le DataSet typé exemple.....	8
Afficher les données.....	9
Le DataSet typé, de plus près	10
Le code de chargement.....	11
La mise à jour des données	13
La mise à jour « manuelle » des données	17
Real Life	18
La base de données	19
Les tables utilisées.....	19
Le but à atteindre	19
L'application	19
Le visuel simplifié	20
La page des clients.....	20
Economie et Pertinence	20
Contexte applicatif et pertinence	20
Le choix retenu pour l'exemple.....	21
La vue de l'onglet « sélection du client »	22
Le code de la page client	22
Les données.....	22
Le DataSet.....	25
La grille	26
Les champs de la fiche.....	27
Les boutons de commande	27
La philosophie de mise à jour.....	28
Le code des boutons.....	29

Le chargement des données	31
Détection des mises à jour	32
La recherche des clients	33
Un peu d'ergonomie	33
La page des commandes	34
La détection du changement de page.....	34
Le chargement des commandes	35
L'affichage des commandes	36
Conclusion	37

Présentation

La logique du « tout déconnecté » proposée il y a déjà longtemps par Borland avec Midas est devenue par la grâce de ADO.NET une méthode incontournable de gestion et d'accès aux données s'opposant au mode connecté appelé plus souvent « client/serveur » lui-même s'étant éloigné inexorablement du mode de gestion dit « réseau » tel qu'on le trouvait sous Access ou Paradox.

Seulement voilà, lorsqu'il s'agit de partager des données et de gérer les accès concurrents, les verrous (locks) *pessimistes* des gestions de bases de données réseau d'il y a 20 ans sont bien plus efficaces du point de vue de l'utilisateur que les verrous dits *optimistes* des serveurs SQL et des middlewares déconnectés !

Difficile en effet d'expliquer à un utilisateur final que les deux heures passées devant une grille de données à corriger des informations sont totalement à refaire parce que, entre temps, quelques utilisateurs ont modifiés des enregistrements de la sélection sur laquelle lui-même travaillait... Le verrou optimiste l'est tellement, optimiste, qu'il en devient stupidement naïf dans la vie réelle où les choses sont loin de se passer aussi benoîtement...

ADO.NET dans la « vraie vie » est donc une autre paire de manches et puisqu'il n'y a plus le choix du mode connecté¹, à nous de développer de nouvelles stratégies pour concilier l'extase optimiste de ADO.NET avec les contraintes de cette vraie vie, plus cruelle, comme les lois de Murphy nous le rappelle tous les jours par leur implacable application à avoir toujours raison au-delà de nos futilités espoirs que tout aille bien...

Une histoire de verrous

Les verrous du mode pessimiste

Il faut en convenir, sans concurrence d'accès la gestion des données serait bien plus simple...

Des solutions à ce problème de partage de l'information avaient déjà fait leur apparition avec les bases de données réseau. Cette architecture frustrante et peu fiable avait néanmoins un énorme avantage : la gestion des conflits d'accès y avait été réglée de la façon la plus radicale qu'il soit : les verrous. Ces verrous qu'on appellera plus tard « pessimistes » permettaient d'éviter à toute personne d'accéder à un enregistrement déjà réservé par un autre utilisateur.

Certes cela n'était pas idéal car certains enregistrements pouvaient rester bloqués des heures durant (plantage machine, lenteur d'un utilisateur, etc) mais cela fonctionnait car les utilisateurs comprenaient facilement cette logique proche de la réalité « papier » : si Albert a pris le dossier du client Durant, alors ce dossier ne peut pas être modifié par Bertrand dans un autre bureau. Si Albert s'endort sur le dossier, Bertrand n'aura d'autre choix que de

¹ Sauf à utiliser ADO.NET à rebrousse-poil, ce qui ne peut que nuire à l'application.

l'appeler ou de se rendre dans son bureau pour rouspéter... Clair et simple à comprendre. Mais pas très « high tech » il faut en convenir !

Une autre situation encore plus risquée avec les verrous est ce qu'on appelle l'étreinte mortelle, ou simplement dead-lock en anglais. Imaginons une ressource A en attente d'une ressource B, elle-même en attente d'une ressource C qui, elle, attend la ressource A... Parfois les chaînes peuvent être très longues car il est difficile, voire impossible, de prévoir toutes ces situations. Les étreintes mortelles demandent le plus souvent l'intervention d'un administrateur qui n'aura pas d'autre choix que de lever tous les verrous ou de « descendre » la base de données pour la « remonter », faisant ainsi perdre le travail de tous les autres utilisateurs.

Les verrous pessimistes étaient pratiques, dans l'esprit, mais peu satisfaisant techniquement parlant, il faut en convenir.

Le mode optimiste sans verrou

Les grandes bases de données SQL étaient utilisées originellement dans des contextes massivement concurrentiels et souvent délocalisés. Dans un tel cadre il n'était pas possible d'autoriser la pose de verrou par chaque utilisateur. On imagine assez bien qu'un système de réservation de places d'avion avec des terminaux dans le monde entier ne puisse pas fonctionner selon ce mode, pas plus aujourd'hui qu'une gestion commerciale avec deux ou trois cents postes clients répartis dans l'entreprise, parfois sur des sites géographiques différents.

De là est née l'idée de la gestion optimiste des conflits d'accès et donc, par opposition le qualificatif de pessimiste à la gestion des verrous discutée plus haut.

En quoi l'optimisme joue-t-il un rôle dans tout ceci ?

Dans le modèle dit pessimiste on suppose par avance qu'il va y avoir un problème (une concurrence d'accès), on se place dans le pire des cas et on s'en prémunit en posant un verrou. Dans le modèle optimiste on suppose au contraire que dans la majorité des cas tout ira bien et qu'il n'est pas nécessaire de poser des verrous, on gèrera les conflits, rares, au cas le cas au moment où l'information sera enregistrée dans la base de données.

Le modèle optimiste est en effet plus proche de la réalité du terrain que le modèle pessimiste : majoritairement aucun conflit n'intervient dans la plupart des applications.

Seulement de là à conclure que les verrous ne sont pas utiles il y a un pas qui a été franchi sans penser au fait que même si les conflits sont rares ils existent malgré tout. Demander à un utilisateur de recommencer son travail dans un tel cas lui cause généralement un désagrément très fort ...

De fait, le modèle optimiste est statistiquement plus proche de la réalité, mais est un véritable désastre du point de vue de la psychologie de l'utilisateur...

Dans un monde se souciant tellement de l'aspect des applications ou de la dernière innovation ergonomique (ou supposée telle) il est curieux de voir à

quel point la frustration de l'utilisateur final est à ce point ignorée dans les gestions de bases de données !

C'est justement entre deux impératifs que le développeur se retrouve coincé : ne pas utiliser le mode pessimiste qui n'est pas fiable, et éviter d'avoir à porter un casque pour se protéger des utilisateurs pas forcément contents de sa vision de l'*optimisme*...

C'est le dernier qui a parlé à raison...

C'est le mode de fonctionnement optimiste par défaut d'une base SQL.

Il est très simple à comprendre et mène à des situations tellement ubuesques que rares sont les applications à l'utiliser².

Pour comprendre illustrons le propos avec une brève séquence :

Originellement supposons une table d'articles dont l'un des enregistrements est le suivant :

Clé	Article	Poids colis (kg)
58	Arrosoir	2,0

Les utilisateurs Alice et Bernard accèdent sans le savoir à ce même article à quelques (milli-) secondes d'intervalle. Chacun possède maintenant une copie de l'article, vraisemblablement dans un DataRow d'un DataSet se trouvant dans la mémoire de chacun de leurs PC.

Alice sachant que les arrosoirs ont été retirés du catalogue va réutiliser l'enregistrement pour un nouvel article. Alice valide sa saisie, la base de données contient désormais l'information suivante :

Clé	Article	Poids colis (kg)
58	Armoire	100,0

Bernard, se faisant le même raisonnement qu'Alice décide de changer le nom de l'article. Il s'agit désormais d'une lampe de bureau. Il ne modifie pas le poids qui par chance est le même que celui de l'enregistrement original qu'il a toujours dans la mémoire de sa machine. Il valide sa modification qui est reportée dans la base de données.

² Ou, quand c'est le cas, nous avons constaté que les développeurs le faisait en ignorant les conséquences de leur choix qui est parfois un « non choix ».

A la prochaine commande de lampes de bureau le service expédition réservera des semi-remorques là où une seule camionnette aurait suffi car l'enregistrement dans la base ressemble maintenant à cela :

Clé	Article	Poids colis (kg)
58	Lampe de bureau	100,0

C'est cela la gestion optimiste des conflits d'accès offerte par défaut par les bases de données SQL.

Il existe toutefois une parade protégeant contre une telle situation. Si celle-ci a pu arriver c'est que la séquence UPDATE de l'application utilisée par Bernard est la suivante :

```
UPDATE TableArticle SET NOM_ARTICLE = ' Lampe de Bureau' WHERE  
CLE=58
```

Si la séquence avait été écrite comme suit, la mise à jour aurait échoué, chagrinant Bernard mais évitant de lourdes pertes à l'entreprise :

```
UPDATE TableArticle SET NOM_ARTICLE = ' Lampe deBureau' WHERE  
CLE=58 AND POIDS_ARTICLE = 2.0 AND NOM_ARTICLE=' Arrosoir'
```

La solution consiste donc à contrôler la totalité des champs (modifiés ou non) dans la clause WHERE pour s'assurer que l'enregistrement qui va être modifié dans la base de données est identique à celui qui a été lu. Si ce n'est pas le cas l'UPDATE échoue et on peut prévenir l'utilisateur en lui proposant, généralement, la nouvelle copie de l'enregistrement. Bien entendu ADO.NET propose par défaut de telles séquences de mise à jour protégées, mais cela ne s'applique pas lorsque c'est le développeur qui saisit lui-même les commandes SQL.

Bien entendu on commence déjà à être pessimiste... Et cela se payera par un alourdissement du développement puisqu'il faudra prévoir le cas où l'UPDATE sera rejeté par la base, faute de trouver un enregistrement correspondant à la clause WHERE.

Rien n'étant parfait, cette solution pose quelques problèmes. Le premier est qu'elle ne fonctionne généralement pas sur les BLOB qui sont de plus en plus utilisés (image ou textes longs de type « mémo »). Le second est qu'elle a un prix non négligeable en termes de performance.

Il existe une stratégie intermédiaire qui consiste à ne tester dans le WHERE que les champs modifiés et la clé primaire.

Dans notre exemple cela aurait évité le problème, mais n'aurait pas interdit d'autres situations gênantes car moins on teste de champs dans le WHERE,

plus grande devient la brèche par laquelle se glissera un conflit d'accès non détecté.

Une autre technique parfois utilisée consiste à utiliser des horodateurs pour contrôler que l'enregistrement en mémoire est le même que celui qui a été lu. Cette technique évite de tester tous les champs dans le WHERE et offre une sécurité presque équivalente (si la précision du champ heure va au moins jusqu'à la milliseconde). Mais on reste bien, il faut l'admettre, plus dans le bricolage que dans la haute technologie...

Mieux vaut prévenir que guérir !

Finalement, en informatique comme en médecine, mieux vaut éviter un risque potentiel que d'avoir à soigner les conséquences d'une situation qui pouvait être prévue. C'est la justification de la prévention, voire du principe de précaution.

Imaginons un peu si du jour au lendemain un champ en friche était transformé en aéroport « sauvage ». Les accidents ne tarderaient pas à être légion. La mauvaise décision consisterait à installer un hôpital à proximité du champ pour soigner le lot quotidien de victimes. La bonne décision serait de baliser les pistes, de construire une tour de contrôle et de rationaliser les déplacements du personnel et des clients pour éviter les accidents.

Avant toute chose, lorsqu'on met en place une gestion de données, il convient donc bien poser le problème et de réfléchir le temps nécessaire à la conception des tables et des relations pour voir comment limiter les conflits d'accès. Ce temps n'est jamais perdu et permet même parfois de concevoir un système sans presque aucune possibilité de conflit.

D'ailleurs certaines règles de bonne pratique sont adoptées par tout le monde, ou presque, sans même avoir pris conscience de leur justification. Le fait d'utiliser comme clé primaire d'une table un champ auto-incrémenté par le serveur en fait partie. L'erreur qui consiste à utiliser un nom de produit ou de client comme clé primaire n'arrive plus (humm !).

Gérer les conflits d'attribution des clés primaires fait partie de la gestion des conflits d'accès. C'est un premier pas, tout petit pour l'informaticien, mais un grand pas pour l'utilisateur !

Les conflits sans transactions

Lorsque les opérations de mises à jour ne concernent qu'un seul enregistrement à la fois nous avons vu que certaines techniques permettaient d'éviter les conflits ou de les gérer en pleine connaissance de cause.

Dans un tel cas il ne semble pas, a priori, nécessaire de gérer des transactions même s'il faut malgré tout noter que certaines bases de données ne peuvent de toute façon travailler que dans le contexte d'une transaction (Interbase et Firebird par exemple).

La logique veut que lorsque plusieurs enregistrements font partie d'un même lot de données à mettre à jour il est fortement conseillé de protéger la totalité de la séquence par une transaction. Ce sujet est débattu dans un précédent article (« Les nouvelles transactions distribuées de .NET 2.0 ») et nous n'en dirons pas plus ici.

Toutefois, il reste possible, selon le contexte applicatif, de se passer d'une transaction. A condition bien entendu de gérer les possibles conflits par programmation.

Le composant `DataAdapter` possède la propriété `ContinueUpdateOnError` qui est à `false` par défaut. Cela signifie que lorsque ADO.NET met à jour les données toute erreur stoppe la séquence et lève une exception.

Comme nous le disons plus haut, dans la grande majorité des cas il est préférable que tout cela soit exécuté dans un contexte transactionnel pour s'assurer que toutes les mises à jour ont été validées. Dans la négative la transaction est annulée (Rollback) et l'utilisateur en est averti.

Néanmoins il reste donc des cas où la mise à jour partielle d'un lot de données peut être envisagé et pris en compte hors contexte transactionnel.

Dans ces situations on place la propriété `ContinueUpdateError` du `DataAdapter` à `true`. De ce fait ce dernier ne s'arrêtera pas en cas d'erreur de mise à jour sur un enregistrement et déroulera toute la séquence jusqu'à son terme sans lever d'exception.

Il faut alors prendre connaissance des éventuelles erreurs pour donner la possibilité à l'application ou à l'utilisateur de corriger la situation. Cela se fait par exemple en interrogeant `DataRow.HasErrors` qui permet, ligne par ligne d'un `DataTable`, de connaître le statut après mise à jour. S'il y a eu un problème durant la mise à jour on peut en savoir la raison en consultant `DataRow.RowError`. De façon similaire le `DataSet` et le `DataTable` possède une propriété `HasErrors` qui permet de prendre connaissance des éventuelles erreurs globalement (toutes lignes ou toutes tables confondues).

Lorsqu'une `DataTable` est connectée à une grille de données, les enregistrements en erreur sont repérés automatiquement par un marqueur visuel, un point d'exclamation. Le tooltip de la ligne donne alors le message d'erreur. Ces mécanismes automatisés de .NET rendent la programmation de cette solution sans transaction très simple à mettre en œuvre.

Une fois les erreurs corrigées, on annule les marqueurs en appelant `ClearErrors` ce qui fonctionne aussi bien au niveau de la ligne `DataRow`, de la table `DataTable` ou de l'ensemble de données `DataSet`.

Les relations maître / détail

Les relations maître / détail sont celles qui font intervenir le fameux « R » des SGBD-R : les relations.

L'un des intérêts majeurs des bases de données est leur capacité à gérer les relations existantes entre les diverses tables qui les composent. Nous n'entrerons pas ici dans les considérations théoriques du relationnel et du modèle entité / relation, le lecteur intéressé pourra se reporter vers notre

article « Les bases de données relationnelles » téléchargeable sur le site de l'auteur³.

Une application bien conçue est une application reposant sur une base de données bien structurée et tirant partie de la puissance des serveurs SGBD-R modernes. Tout cela implique des relations, des clés primaires et étrangères, un choix judicieux des numéricités, etc.

Néanmoins il existe un certain fossé entre le modèle entité / relation des bases de données et la façon dont les applications accèdent aux données. ADO.NET, notamment par le DataSet capable de stocker tables et relations, apporte un ensemble de dispositions aidant à une exploitation rationnelle des données. Mais c'est sans prendre en compte qu'entre lire des données et les mettre à jour il y a distance importante...

Aujourd'hui toute base de données sait retourner des informations en provenance de plusieurs tables simplement en utilisant la notion de jointure dans un ordre SELECT. Mais aucune ne propose quelque chose d'aussi élégant quand vient le moment de modifier, insérer ou supprimer des informations dans un tel ensemble de données composite.

Par exemple, les vues modifiables, bien que partie intégrante de la norme SQL⁴, ne sont pas ou que partiellement implémentées par les serveurs. Ainsi, la difficile tâche de mettre à jour des informations relationnelles revient uniquement au développeur.

Si l'on ajoute à cette difficulté celle d'un environnement totalement déconnecté comme ADO.NET, on voit clairement qu'il y a matière à réflexion pour dégager une stratégie fiable.

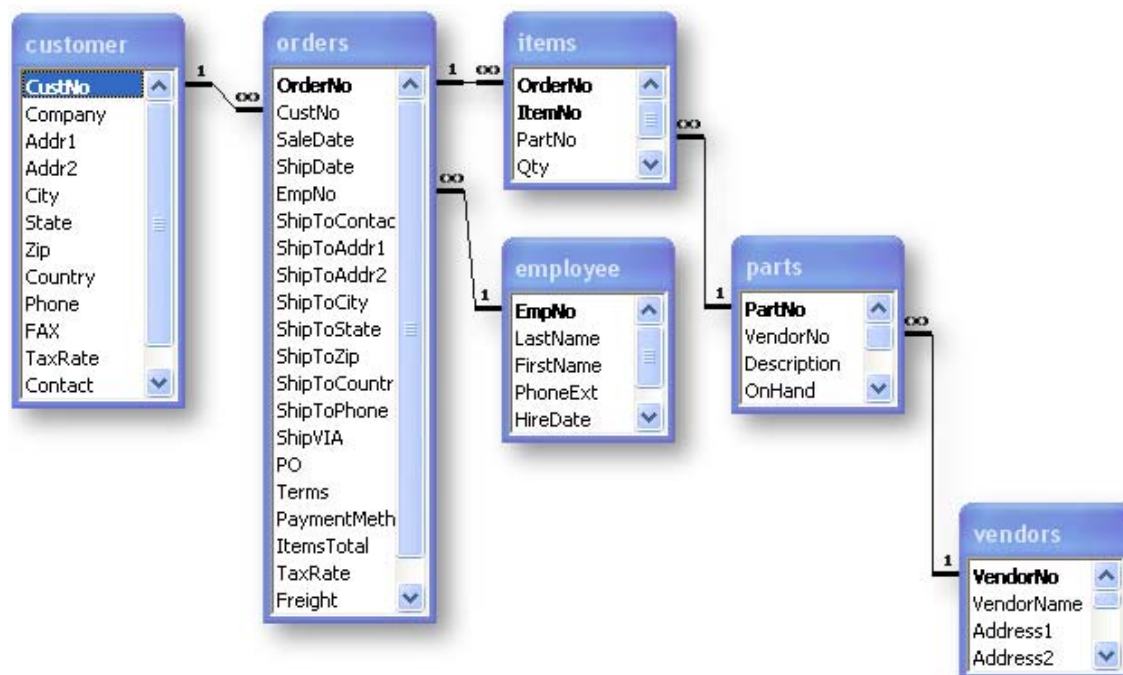
La base exemple

Dans les propos qui vont suivre nous avons choisi une base de données exemple simple à comprendre. Les utilisateurs de Delphi la connaissent bien, il s'agit de DBDEMOS fourni avec ce langage depuis toujours sous différents formats (Interbase, Access, Paradox, ...). C'est la version Access que nous utiliserons ici. Certes, Access n'est pas un SGBD-R comme Oracle ou SQL Server, bases qui sont le véritable propos de cet article, mais Access est suffisamment puissant et universellement connu pour servir les exemples de code qui viendront. Tout le monde a ou peut avoir le moteur JET sur sa machine, imposer au lecteur d'installer SQL Server ou Oracle ne semblait pas une option judicieuse.

Le schéma de la base de test est le suivant :

³ Rubrique téléchargement, groupe « BorCon 2003 » depuis www.e-naxos.com.

⁴ SQL-92 définit les vues modifiables avec malgré tout des restrictions importantes et rares sont les bases de données à implémenter ne serait-ce que les exigences de cette norme.



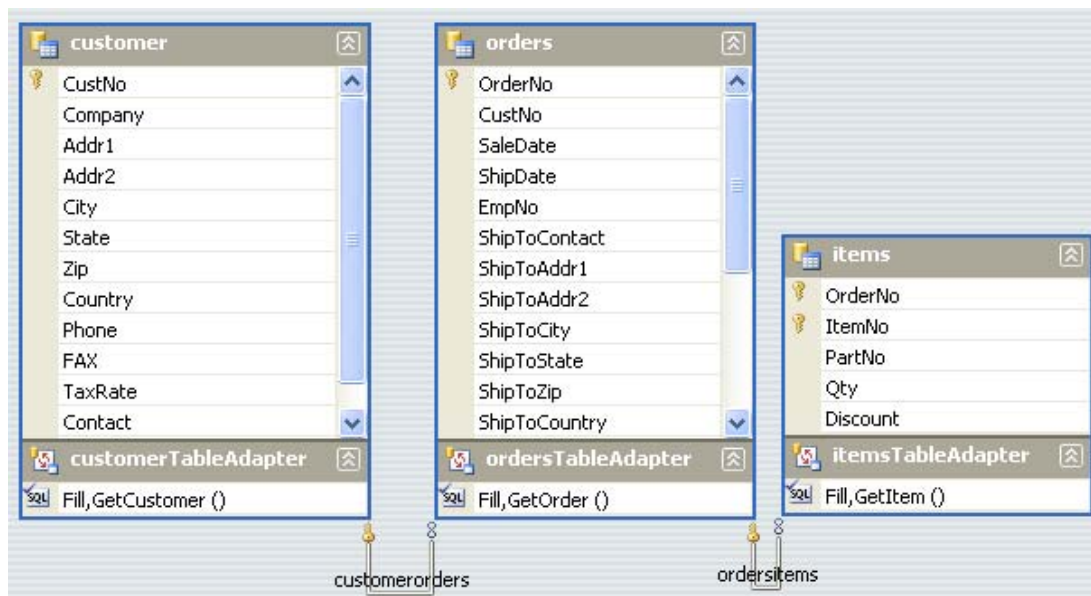
On trouve :

- Customer, la table des clients ;
- Orders, la table des entêtes de commandes ;
- Items, la liste des articles commandés dans une commande ;
- Employee, les employés ayant servis une commande ;
- Parts, le catalogue des articles ;
- Vendors, la table des fournisseurs.

Ce contexte applicatif est un cas d'école et nul besoin de plus le commenter, tout le monde sait comment une gestion de commande est architecturée et quelles sont ces buts. De plus nous n'utiliserons pas forcément toutes les tables ici montrées.

Le DataSet typé exemple

Pour simplifier l'exemple et parce qu'il s'agit là d'une méthode de travail recommandée (à juste titre), nous allons utiliser un DataSet typé. Il reprendra ici une partie de la base exemple. En voici le schéma sous Visual Studio 2005 dans le concepteur visuel des ensemble de données typés :



On retrouve la table des clients (customer), celle des commandes (orders) et enfin celle des lignes de commande (items). Dans cet exemple nous nous contenterons d'afficher le numéro d'article (PartNo) et non sa désignation pour éviter d'utiliser une quatrième table (Parts).

On remarquera les deux relations « customersorders » et « ordersitems » permettant à l'expert créateur de DataSet typés de VS 2005 de construire toutes les classes représentant les tables ainsi que leurs relations. Nous en ferons usage dans le code qui va suivre.

Afficher les données

Gérer l'affichage des données n'est pas le sujet de cet article, mais il faut bien planter le décor...

La fiche⁵ sera la plus simple possible, composée de trois grilles DataGridView et d'un composant OleDbConnection qui assurera la connexion à la base DBDEMOS.mdb. Le tout sera agrémenté de deux boutons, l'un permettant de quitter l'application, l'autre de mettre à jour les données après leur modification.

Pour l'instant nous n'avons pas encore codé la mise à jour. Concentrons-nous sur l'affichage. En activité la fiche ressemble à cela :

⁵ L'application fournie avec l'article s'appelle « RealLife1 »

Clients (customer)

CustNo	Company	Addr1	Addr2	City	State
1221	Kauai Dive Shoppe	4-976 Sugarloaf ...	Suite 103	Kapaa Kauai	HI
1231	Unisco	PO Box Z-547		Freeport	
1351	Sight Diver	1 Neptune Lane		Kato Paphos	
1354	Cayman Divers ...	PO Box 541		Grand Cayman	
1356	Tom Sawyer Divi...	632-1 Third Fry		Christiansted	St. Croix

Commandes (orders)

OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToContact
1003	1351	12/04/1988	03/05/1988	114	
1052	1351	06/01/1989	07/01/1989	144	
1055	1351	04/02/1989	05/02/1989	29	
1067	1351	01/04/1989	02/04/1989		
1075	1351	21/04/1989	22/04/1989		

Lignes (items)

OrderNo	ItemNo	PartNo	Qty	Discount
1067	1	12317	5	0

Buttons: MAJ, Quitter

Depuis le concepteur visuel nous n'avons fait que placer les grilles sans modifier leurs propriétés et sans leur assigner de source de données, tout sera fait par code pour mieux illustrer le propos. Seul le composant de connexion a été paramétré pour accéder à la base Access.

Le DataSet typé, de plus près

Dès lors qu'un DataSet typé a été créé les choses sont très simples puisque VS 2005 génère automatiquement tout le code nécessaire à la gestion des tables et des relations. Mais qu'est-ce qui est réellement généré et comment s'en servir ?

La classe de notre DataSet s'appelle DbDemos, nous l'avons créée en ajoutant un nouvel item à notre projet et en choisissant « DataSet ». Les trois tables ont été posées par drag and drop depuis l'explorateur de base de données qui contient la connexion OleDb que nous avons déjà créée (un composant OleDbConnection).

Le DataSet typé est représenté par plusieurs codes sources, notamment son schéma XSD qui ici s'appelle donc DbDemos.xsd. Ce fichier est représenté par un schéma visuel lorsqu'on double-clique dessus. VS a créé pour nous plusieurs autres fichiers, mais celui qui mérite une petite visite s'appelle DbDemos.Designer.cs. C'est lui qui contient le code C# de toutes les classes. Celle du DataSet lui-même et celles de ses tables.

Notre projet s'appelant « RealLife1 », ces définitions sont placées dans l'espace de noms RealLife1. S'agissant de l'espace de noms par défaut du projet nous n'avons rien à faire pour utiliser le DataSet dans notre application de test.

Toutefois, à y regarder de plus près dans ce très long code de plus de 4500 lignes, on s'aperçoit qu'un second espace de noms s'y trouve défini... Il s'agit de Real Li fe1. DbDemosTabl eAdapters. Pour accéder aux classes qu'il définit il est donc pratique d'ajouter dans la fiche de notre exemple un `using`. Car ce qui se cache dans cette seconde partie du code source généré est essentiel : il s'agit de tous les DataAdapters spécialisés qui ont été écrits par VS pour chacune des tables de notre DataSet. Tout y est, pour chaque table, et surtout les quatre commandes SELECT, INSERT, UPDATE, DELETE avec le code SQL nécessaire à leur exécution. Tout ce code va nous simplifier la programmation et il est dommage que si peu de gens en connaissent l'existence.

Le code de chargement

Dans la majorité des cas les données sont affichées dès qu'une fiche est ouverte. C'est donc dans l'événement `Form_Load` que se trouvera le plus généralement la séquence de chargement des données.

Toutefois il est plus judicieux de placer ce code dans une méthode séparée et d'appeler cette dernière dans le `Form_Load`. C'est ce que nous allons faire en créant une méthode `LoadData()` dans la fiche `Form1` tel que montré plus bas.

```
private void LoadData ()
{
    // Création du DataSet typé
    DbDemos db = new DbDemos();
    // Création de l'adapter pour la table customer
    customerTableAdapter c_da = new customerTableAdapter();
    c_da.Connection = DbDemosConnection; // connexion
    c_da.Fill (db.customer);             // remplissage
    // même séquence pour la table orders
    ordersTableAdapter o_da = new ordersTableAdapter();
    o_da.Connection = DbDemosConnection;
    o_da.Fill (db.orders);
    // même séquence pour la table items
    itemsTableAdapter i_da = new itemsTableAdapter();
    i_da.Connection = DbDemosConnection;
    i_da.Fill (db.items);
    // On connecte la grille customer au DataSet
    CustomersGrid.DataSource = db;
    // puis on indique son membre
    CustomersGrid.DataMember = "customer";
    // même séquence pour la grille orders
    OrdersGrid.DataSource = db;
    OrdersGrid.DataMember = "customer.customerorders";
    // même séquence pour la grille items
    ItemsGrid.DataSource = db;
    ItemsGrid.DataMember = "customer.customerorders.ordersitems";
}
```

Ce code n'a rien d'exceptionnel, mais certaines choses méritent d'être commentées :

- Le DataSet est créé dans la méthode, il n'est donc pas accessible directement en dehors de celle-ci. De même, chaque appel à la méthode créera un nouveau DataSet, l'ancien sera libéré par .NET. Cette façon de faire n'est pas forcément la mieux adaptée si le reste du code de la fiche doit faire référence au DataSet. Mais cela sera suffisant pour notre exemple.
- Le DataSet qui est créé l'est depuis la classe du DataSet typé que nous avons créé, DbDemos. C'est un point important de la démonstration.
- Pour chaque table à remplir avec les données nous utilisons non pas un simple DataAdapter mais bien ceux qui ont été générés par VS en même temps que le DataSet typé. Par exemple pour la table des entêtes de commandes (orders) nous utilisons ordersTableAdapter. Cela est aussi essentiel dans cette démonstration puisque nous évitons ainsi d'avoir à saisir le code SQL. Nous travaillons et raisonnons Objet et non plus SGBD.
- On remarquera aussi que l'appel à la méthode Fill () des DataAdapters s'effectue en passant en paramètre les objets tables typés, par exemple db.orders. Nous ne passons pas le DataSet directement, nous n'utilisons pas le TableMappings des DataAdapters notamment ni d'autres techniques similaires du DataSet. Aucune constante chaîne n'est utilisée, ni code SQL, tout le travail a été fait par VS dans le DataSet typé.
- La connexion des grilles de données est un peu plus subtile. En effet, nous commençons par connecter le DataSet à la propriété DataSource de la grille, ce qui est conventionnel, puis nous indiquons un DataMember sous la forme d'une chaîne de caractères... Et comme vous le constatez ces noms semblent bien étranges. Nous allons en discuter plus bas.
- Enfin les DataAdapters chargent les trois tables en se basant sur le schéma que nous avons créé lors de la conception de la classe du DataSet typé. Tel que nous avons procédé c'est bien la totalité de ces tables qui sera chargé en mémoire. Cela convient parfaitement pour un exemple et certainement dans quelques cas réels, mais il semble définitivement déraisonnable de procéder de la sorte avec des tables de plusieurs milliers ou millions d'enregistrements. Dans de tels cas il sera nécessaire de filtrer les tables en ajoutant des paramètres par un WHERE dans les ordres de chargement. Nous verrons cela à l'œuvre dans la seconde partie de cet article, section appelée à juste titre « Real Life ».

Ce qui nous intéresse ici c'est d'afficher les trois tables, chacune dans une grille, et que ces trois grilles soient synchronisées afin de refléter les relations qui existent dans la base de données.

La première étape, la plus importante, a été de concevoir un DataSet typé qui tient compte des relations, ce que nous avons fait.

La seconde étape cruciale pour obtenir l'effet rechercher consiste à connecter les grilles aux bonnes sources de données. C'est là qu'interviennent les chaînes de caractères utilisées pour spécifier les DataMembers des grilles.

La grille des clients utilise la chaîne "customer". C'est tout simplement le nom de la table dans le DataSet.

La grille des entêtes de commandes utilise la chaîne "customer.customerorders". D'où vient cette chaîne ?

Cela est très simple : ce qui nous intéresse ici ce n'est pas le « bête » contenu de la table des entêtes de commandes, mais uniquement les entêtes de commandes qui correspondent au client en cours de sélection. Or, si nous spécifions pour la grille une DataSource de type DbDemos.orders, c'est-à-dire la DataTable typée contenue dans le DataSet, nous obtiendrons l'affichage de la totalité des informations de cette table, sans aucun filtrage. Quelque soit le client sélectionner nous aurons dans la grille des commandes toutes celles qui existent. Ce n'est pas du tout l'effet recherché.

Si vous revenez sur le schéma du DataSet que nous avons conçu (plus haut dans cet article), vous remarquerez que les relations portent des noms. Celle qui relie la table des clients aux entêtes de commande s'appelle customerorders, nom créé automatiquement par VS en fonction du nom des tables liées et que nous aurions pu (dû) modifier pour le rendre peut-être plus parlant. En tout étant de cause ce nom sur le schéma n'a pas qu'un but décoratif ou informatif... VS va s'en servir pour nommer une propriété de la table des clients, propriété qui représente la relation et ainsi les lignes d'entêtes de commandes relatives au client en cours.

La grille de données des entêtes de commande n'est donc pas reliée directement à la DataTable des entêtes de commandes, mais à la propriété customerorders de la table customer, ce qui s'écrit « customer.customerorders » selon une syntaxe en fait très habituelle.

La même explication s'applique à la grille des lignes de commandes. Dans ce cas la chaîne membre accède à la seconde relation et cela à partir de la précédente. Le résultat souhaité étant d'obtenir les lignes qui correspondent à l'entête de commande sélectionnée parmi celles du client sélectionné. D'où la chaîne "customer.customerorders.ordersitems", ordersitems étant une propriété de la table des entêtes de commandes, tout comme customerorders est une propriété de la table customer.

Tout s'explique...

La mise à jour des données

Une fois les grilles connectées convenablement et les données chargées nous disposons d'une base solide pour passer à l'étape suivante, celle de la mise à jour des données.

Une fois encore nous mettrons à profit le DataSet typé que nous avons créé ainsi que le DataAdapters que VS a générés pour nous.

Le code de chargement réclame un petit changement : pour accéder au DataSet depuis la méthode UpdateData() qui sera chargée de la mise à jour il faut que l'instance soit déclarée dans la fiche et non pas dans la méthode LoadData(). Une fois cet ajustement effectué nous pouvons écrire la séquence de sauvegarde.

En réalité nous allons vous montrer deux séquences. La première est la meilleure car dans le contexte d'un ensemble de données relationnelles il est impératif de protéger la mise à jour en l'englobant dans une transaction. La seconde séquence ne prend pas en charge la transaction car notre exemple utilise Access qui ne supporte pas les transactions.

The screenshot shows a Windows application window titled "Form1" with three data grids and three buttons.

Top Grid (Customers):

	CustNo	Company	Addr1	Addr2	City	State
	1221	Kauai Dive Shoppe	4-976 Sugarloaf ...	Suite 103	Kapaa Kauai	HI
	1231	Unisco	PO Box Z-547		Freeport	
	1351	Sight Diver	1 Neptune Lane		Kato Paphos	
▶	1354	Cayman Divers ...	PO Box 541		Grand Cayman	
	1356	Tom Sawyer Divi...	632-1 Third Fryde...		Christiansted	St. Croix

Middle Grid (Orders):

	OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToContact
	1104	1354	18/07/1992	18/07/1992	83	
	1292	1354	01/01/1995	01/01/1995	136	
▶*		1354				

Bottom Grid (Order Items):

	OrderNo	ItemNo	PartNo	Qty	Discount
*					

Buttons: MAJ, Recharger, Quitter

La copie d'écran ci-dessus montre les trois grilles durant l'insertion d'une nouvelle commande pour le client 1354. Dans la structure de la base de données le numéro de commande n'est pas automatique, il faut donc le saisir (c'est un cas particulier comme un autre), en revanche ADO.NET remplit automatiquement le numéro de client car il connaît la relation reliant les lignes de commandes aux clients. Ce mécanisme automatisé garantit l'intégrité référentielle lors de la saisie de nouvelles données.

Bien entendu, dans la réalité nous aurions caché la colonne CustNo dans les entêtes de commandes, cela n'a pas d'intérêt pour l'utilisateur, d'autant que normalement le client en cours est affiché dans la première grille.

Rappelons que pour l'instant le travail que nous effectuons se trouve uniquement dans la mémoire de notre machine.

L'image suivante montre, une fois l'entête de commande créée, la saisie d'une ligne article. Ici aussi nous voyons que ADO.NET place automatiquement le numéro de commande, 9999, dans le champ OrderNo qui lie la table items à la table orders par la relation existante dans la base de données que nous avons reprise lors de la modélisation du DataSet typé.

The screenshot shows a Windows application window titled 'Form1' with three data grids and three buttons on the right.

Grid 1 (Customers):

	CustNo	Company	Addr1	Addr2	City	State
	1221	Kauai Dive Shoppe	4-976 Sugarloaf ...	Suite 103	Kapaa Kauai	HI
	1231	Unisco	PO Box Z-547		Freeport	
	1351	Sight Diver	1 Neptune Lane		Kato Paphos	
▶	1354	Cayman Divers ...	PO Box 541		Grand Cayman	
	1356	Tom Sawyer Divi...	632-1 Third Fryde...		Christiansted	St. Croix

Grid 2 (Orders):

	OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToContact
	1104	1354	18/07/1992	18/07/1992	83	
	1292	1354	01/01/1995	01/01/1995	136	
▶	9999	1354	01/01/2007	15/02/2004	83	
*						

Grid 3 (Items):

	OrderNo	ItemNo	PartNo	Qty	Discount
	9999	1	1946	2	10
✎	9999	2	2367	1	5
*					

Buttons:

- MAJ
- Recharger
- Quitter

Le code du bouton de mise à jour est le suivant dans la version Access qui ne gère pas les transactions :

```
private void UpdateData()
{
    customerTableAdapter c_da =
        new customerTableAdapter();
    c_da.Connection = DbDemosConnection;
    c_da.Update(db.customer);
    ordersTableAdapter o_da =
        new ordersTableAdapter();
    o_da.Connection = DbDemosConnection;
    o_da.Update(db.orders);
    itemsTableAdapter i_da =
        new itemsTableAdapter();
    i_da.Connection = DbDemosConnection;
    i_da.Update(db.items);
}
```

Comme on peut le constater ce code est très simple. Une bonne raison à cela, nous ne faisons que réutiliser les trois DataAdapters qui ont été créés automatiquement avec le DataSet typé par l'expert de Visual Studio... Ces classes contiennent toute les séquences SQL nécessaire à l'insertion, la suppression et la modification des tables correspondantes, et tout cela sans rien avoir fait d'autre qu'un joli dessin (le schéma du DataSet typé) !

La séquence utilisée se compose ainsi de trois sections identiques, chacune étant formée par :

- la création d'une instance de l'adaptateur de la table à mettre à jour
- L'appel à la méthode Update() de ce dernier en passant en paramètre la DataTable typée du DataSet typé.

On remarquera que la séquence respecte un ordre particulier et que les trois tables sont traitées en partant de celle placée le plus haut dans l'ordre des relations maître / détail jusqu'à la table « feuille » de cet arbre. Ainsi nous mettons d'abord à jour les clients, puis les entêtes de commandes, et enfin les lignes de commande. Procéder dans un ordre différent ne fonctionnerait pas vis-à-vis de la base de données. Rappelons-nous en effet que toutes les données sont dans la mémoire de l'application. De fait il est impossible d'ajouter dans la base de données des lignes à une commande qui n'existe pas déjà dans cette dernière.

Notre séquence fonctionne bien et remplit son but, c'est-à-dire synchroniser la base de données avec sa représentation dans la mémoire de l'application.

Malgré tout il reste un détail qui n'a pas été traité : la transaction.

Dans un contexte relationnel il est indispensable de s'assurer que toutes les mises à jour sont acceptées ou rejetées en bloc afin de garantir l'intégrité des informations.

Comme nous le disions, la base Access utilisée en exemple ne supporte pas les transactions, c'est la raison pour laquelle on préférera dans la réalité utiliser une « vraie » base de données comme SQL Server par exemple.

Ainsi, pour toute autre base de données, la séquence proposée plus haut ne sera pas appelée directement, mais depuis une autre méthode qui protégera l'ensemble de la mise à jour par une transaction. Ce code est donné ici :

```
// Mise à jour des données
private void SaveData()
{
    try
    {
        using (TransactionScope scope =
            new TransactionScope())
        {
            UpdateData(); // la méthode sans trans.
            scope.Complete();
        }
    }
    catch(TransactionAbortedException)
    {
        MessageBox.Show("La mise à jour a échoué !");
    }
}
```

Le code ci-dessus réutilise la méthode UpdateData() vue plus haut mais en la protégeant dans une transaction. Cette dernière n'est pas directement une transaction base de données mais une transaction .NET 2.0 (voir notre article à ce sujet déjà cité dans le présent document).

On remarquera que ce code gère l'exception qui peut se produire si jamais la transaction échoue. Ici nous nous contentons d'un simple message à l'utilisateur car c'est souvent le maximum qu'on puisse faire... Selon le contexte le traitement d'erreur peut se poursuivre par un rechargement des données afin que l'utilisateur puisse repartir sur une base saine (nous verrons cela dans la seconde partie de l'article).

La mise à jour de données relationnelles n'est donc pas quelque chose de très complexe dès lors qu'on utilise correctement les méthodes mises à notre disposition par Visual Studio, notamment la création de DataSet typés.

La mise à jour « manuelle » des données

Dans certains cas on préférera écrire le code SQL de mise à jour et exécuter celui-ci par le biais d'objet DbCommand. Dans ce cas particulier il faut veiller à passer les trois commande UPDATE, DELETE et INSERT dans un ordre logique pour la base : d'abord les enregistrements ajoutés (INSERT), ensuite les enregistrements modifiés (UPDATE) et enfin seulement les enregistrements supprimés (DELETE), le tout dans une logique transactionnelle en tout point identique à ce qui a été vu ici.

Une question se pose alors : comment séparer ces trois ensembles de données dans chaque table ?

Deux approches sont possibles, la première qui vient généralement à l'esprit consiste à balayer chaque table et, en fonction du RowState de chacune, d'envoyer la bonne commande. Mais comme il faut respecter l'ordre indiqué plus haut cela oblige donc pour chaque table à trois balayages successifs (un pour les INSERT, un pour les UPDATE et un autre pour les DELETE).

Si cela est parfaitement réalisable et atteindrait l'objectif fixé, il convient de constater qu'il s'agit là d'une programmation lourde et pour le moins inélégante.

ADO.NET met à notre disposition d'autres moyens plus évolués.

En effet, les objets tables offrent une méthode `Select()` existant en plusieurs versions surchargées et permettant de filtrer et trier les données le tout en récupérant un tableau de `DataRow`. Or il s'avère que cette méthode existe aussi dans une variante acceptant un paramètre de type `DataViewRowState`. Ce paramètre permet de filtrer le résultat obtenu en fonction de l'état de chaque ligne de la table sous-jacente et notamment d'obtenir les lignes ajoutées, supprimées ou modifiées.

Ainsi, pour récupérer uniquement les lignes ajoutées dans la table des commandes nous pouvons utiliser le code suivant :

```
db.customer.Select(' ', ' ', DataViewRowState.Added)
```

L'ensemble de lignes retournées peut être directement passé à la méthode `Fill()` d'un adaptateur pour lequel nous aurons écrit nous-mêmes la commande INSERT.

On fera de même pour les lignes supprimées ou modifiées.

Cette méthode n'est certes pas la plus simple et dès lors qu'on utilise des `DataSet` fortement typés il semble inutile de vouloir réinventer la roue. Mais chaque application et chaque contexte applicatif sont différents et dans certains cas il peut, pourquoi pas, s'avérer plus simple, plus rapide, plus efficace de traiter les données manuellement plutôt qu'en passant par les automatismes proposés.

Real Life

Toutes les techniques discutées jusqu'ici sont essentielles et permettent de mieux comprendre comment tirer partie de ADO.NET et des bases de données sous-jacentes.

Mais en réalité, et même si nous sommes déjà allés beaucoup loin dans le réalisme que certains ouvrages parus sur le sujet, l'obligation de simplification qui préside à tout discours général nous a éloignés un peu de la pratique dans la vie réelle.

Un simple exemple : dans une application sérieuse personne ne chargera la totalité de trois tables ou plus dans un `DataSet`, ce n'est tout simplement pas raisonnable. Tout ce que nous avons pu lire sur le sujet s'arrête pourtant là. C'est dommage car le plus dur reste à faire : passer de la théorie à une application réelle pour de vrais utilisateurs...

Dans cette partie de l'article l'auteur va s'efforcer de créer une application qui dans ses mécanismes répond à la réalité du terrain. Et vous verrez que malgré les connaissances acquises dans la première partie de ce document, il y reste beaucoup de choses à inventer pour atteindre l'objectif.

La base de données

Nous conserverons la base Access exemple utilisée jusqu'ici et pour les mêmes raisons (tout le monde peut utiliser le fichier MDB fourni avec l'article alors qu'il en irait autrement d'une base SQL spécifique).

Les schémas publiés dans la première partie de l'article restent donc d'actualité.

Rappelons qu'une base Access n'a besoin que du moteur JET, généralement installé sur la plupart des machines Windows.

Les tables utilisées

Pour coller à l'esprit de notre démonstration (passer de la théorie à la réalité) nous utiliserons les trois mêmes tables de base (clients, commandes et lignes de commande).

Nous ferons intervenir éventuellement d'autres tables du modèle, comme celle des pays.

Le but à atteindre

En repartant justement des mêmes trois tables il est intéressant de ce demander comment nous réaliserions un véritable logiciel intégrant les contraintes de la vie réelle.

Bien entendu nous n'allons pas nous lancer dans la réalisation complète d'une application de gestion commerciale, cela sortirait malgré tout totalement du cadre d'un article (qui devient déjà bien long !).

En revanche nous nous efforcerons de mimer l'ensemble des contraintes techniques spécifiquement liées à l'exploitation des données, ce qui est le sujet de cet article.

L'application

Partant des tables choisies, la logique d'une application réelle serait la suivante :

- Affichage d'une liste de client pour faire un choix
- Affichage du détail de la fiche sélectionnée
- Possibilité d'édition des champs de cette fiche
- Affichage de la liste des commandes du client sélectionné
- Possibilité de créer une nouvelle commande
- Affichage des lignes de la commande sélectionnée
- Possibilité d'éditer les lignes ou d'en ajouter de nouvelles

Un tel plan est déjà ambitieux. Il nous faudra simplifier l'interface et certaines possibilités pour rester dans le cahier des charges de l'article.

Par exemple, la visualisation / modification du détail d'une fiche client se fera sur quelques champs au lieu de la totalité. En « grignotant » ainsi sur les détails nous pourrions conserver le plus important : la logique et l'architecture.

Le visuel simplifié

Nous utiliserons une fiche Windows Forms sur laquelle nous placerons un contrôle à onglets. La première page concernera la sélection et l'affichage du client, le second la liste des commandes et le troisième le détail d'une commande.

La page des clients

Entrons dans le vif du sujet.

Nous disposons d'un DataSet typé, celui de la première partie de l'article, c'est lui que nous ré-exploiterons. Pour rappel il contient les tables customer, orders et items ainsi que les relations les unissant.

Afficher une grille de données est très pratique et permet une sélection rapide d'un élément parmi plusieurs. Mais dans la réalité il n'est pas question d'afficher la liste de tous les clients contenus dans la base de données ! A cela deux bonnes raisons. La première est d'ordre technique, si le fichier contient des milliers ou des millions de clients nous saturerons très vite la mémoire du PC client, parallèlement nous aurons stressé la base de données inutilement et enfin nous aurons engorgé le réseau. Cela fait beaucoup ! Cela force à l'*économie*. La seconde raison est d'ordre humain. Soyons réalistes, aucun humain ne lira plus d'une vingtaine de ligne dans une grille, voire deux ou trois pages de dix à vingt lignes mais certainement pas 100 ou 2000 pages d'un fichier client. Cela force donc à la *pertinence*...

Economie et Pertinence

S'il avait été informaticien l'excellent Devos aurait pu nous décliner ces deux mots avec sa verve et sa malice. L'économie oblige à la pertinence et la pertinence induit l'économie. Entendez-vous l'écho de l'économie dans la pertinence ? Economie et Pertinence, EP, Êtes-vous Pertinent ? PE, Pouvez-vous être Économe ? etc...

Plus prosaïquement dans notre cas cela signifie que d'une part nous ne devons remonter de la base de données qu'un sous-ensemble de la table des clients (*principe d'économie*) et que ce sous-ensemble de données doit être bien choisi pour que l'utilisateur y trouve rapidement la fiche qu'il cherche (*principe de pertinence*).

Contexte applicatif et pertinence

C'est le contexte applicatif qui doit être le seul guide du développeur pour le choix de la pertinence. En réalité, sauf cas exceptionnels qui ont une solide justification fonctionnelle, les grilles de données dépassant une quarantaine d'enregistrements ne sont que l'éclatante manifestation d'un échec : celui d'une analyse faible et d'une méconnaissance du métier de l'utilisateur.

La logique est finalement la suivante : « puisque je ne sais pas quelles données sont pertinentes alors je les affiche toutes » !

Pour en revenir à la table des clients, il faut ainsi se poser la question de savoir qui sont les utilisateurs de telle ou telle autre fiche de l'application puis comprendre comment la secrétaire, le comptable ou le commercial vont chercher une fiche et selon quels critères. Cette connaissance de l'utilisateur et de ses méthodes de travail est essentielle.

Le comptable, habitué aux chiffres et travaillant sur des pièces comportant les codes (factures, commandes, relevés et relances...) aura peut-être pour habitude de chercher par code client. Alors que la secrétaire ou la standardiste qui auront un « monsieur Durand » au téléphone de la « société Truc » chercheront dans la majorité des cas sur ces deux critères (nom du contact, nom de la société). Dans certains cas ces personnes auront même besoin d'une recherche par approximation phonétique de type Soundex. En effet, au téléphone il est indélicat de demander à une personne de répéter son nom de famille et certains n'ont pas une graphie évidente, il est alors primordial de retrouver rapidement la fiche d'un monsieur Dijkstra alors qu'on aura entendu et tapé Dijta ou Discra (le client attend en écoutant la même boucle de 20 secondes de Vivaldi, et ça énerve, ne l'oubliez pas !). Enfin, le commercial lui, utilisera peut-être un mélange de toutes ces techniques de recherche plus d'autres comme la recherche par ville, secteur géographique, pays, chiffre d'affaire, date de la dernière commande, etc.

Comme on le voit nettement ici la notion de pertinence ne connaît pas d'absolu, c'est le contexte applicatif et la connaissance du métier des utilisateurs qui seuls peuvent prétendre guider le développeur.

Si lors de la réalisation d'un logiciel vous ne disposez pas de ces éléments, c'est que l'analyste n'a pas fait son travail jusqu'au bout.

Le choix retenu pour l'exemple

Bien entendu pour notre exemple nous n'avons pas d'utilisateur à interroger, nous allons ainsi choisir une solution simple à implémenter pour ne pas trop nous écarter de l'essentiel.

De fait nous permettrons à l'utilisateur de sélectionner les clients en fonction du nom de la société.

Cela implique plusieurs choses :

- fournir un champ d'édition pour la saisie du nom recherché
- autoriser la saisie d'un nom partiel
- charger uniquement les données répondant au critère
- afficher pour information le nombre de fiche retournées

La vue de l'onglet « sélection du client »

The screenshot shows a Windows application window titled "Form1". It has three tabs: "Client", "Commandes", and "Lignes". The "Client" tab is active. Inside the tab, there is a search bar containing the letter "A", a "Chercher" button, and a label "label5". Below this is a table with four columns: "Société", "Code", "Ville", and "Contact". The table body is currently empty. At the bottom of the form, there are four input fields: "Société", "Pays" (with a dropdown arrow), "Téléphone", and "Contact". At the very bottom are two buttons: "Annuler" and "MAJ Client".

La copie d'écran ci-dessous montre l'onglet « Client » ouvert.

Il se compose d'un textbox contenant la lettre « A », d'un bouton « Chercher », d'un label pour indiquer le nombre d'enregistrements retournés, d'une grille montrant quatre champs principaux de la table, d'une zone de saisie pour modifier ou créer un client. Enfin, deux boutons ponctuent le bas de page.

Le code de la page client

Nous allons présenter le code en découpant les besoins par grandes fonctions.

Les données

Nous réutilisons ici le DataSet typé des exemples de la première partie de l'article, la classe DbDemos. Toutefois ici nous n'allons plus charger l'ensemble des trois tables d'un seul coup, mais au fur et à mesure des besoins et en limitant la quantité de données remontées.

Comme vous le voyez sur l'interface, nous avons placé un textbox et un bouton « chercher », le textbox étant pré-rempli à la conception avec la lettre « A ».

Notre idée ici c'est de permettre à l'utilisateur de taper les premières lettres d'un nom de société pour ne charger que la liste de celles qui correspondent à sa saisie. Mais lors du premier affichage, que faire ?

Deux écoles s'affrontent : ceux qui préfèrent afficher quelque chose plutôt qu'un écran vide, et ceux qui vont jusqu'au bout de la démarche de pertinence et qui n'affiche rien jusqu'à ce que l'utilisateur choisisse lui-même de faire une recherche.

Cette dernière option a notre préférence, elle est cohérente. Pour les besoins de l'exemple nous opterons pour la première solution, cela ne change guère le code de toute façon. De plus, le pré-chargement de données à l'entrée d'une fiche n'est pas forcément une erreur, dans certains cas cela est plus proche de la réalité notamment quand on peut prévoir quelles données sont, a priori, pertinentes. Une liste d'articles de presse peut être pré-chargée sur les articles du jour, ou bien sur un tri statistique de ceux les plus consultés, etc. Encore et toujours c'est le contexte fonctionnel qui donne les clés de la pertinence...

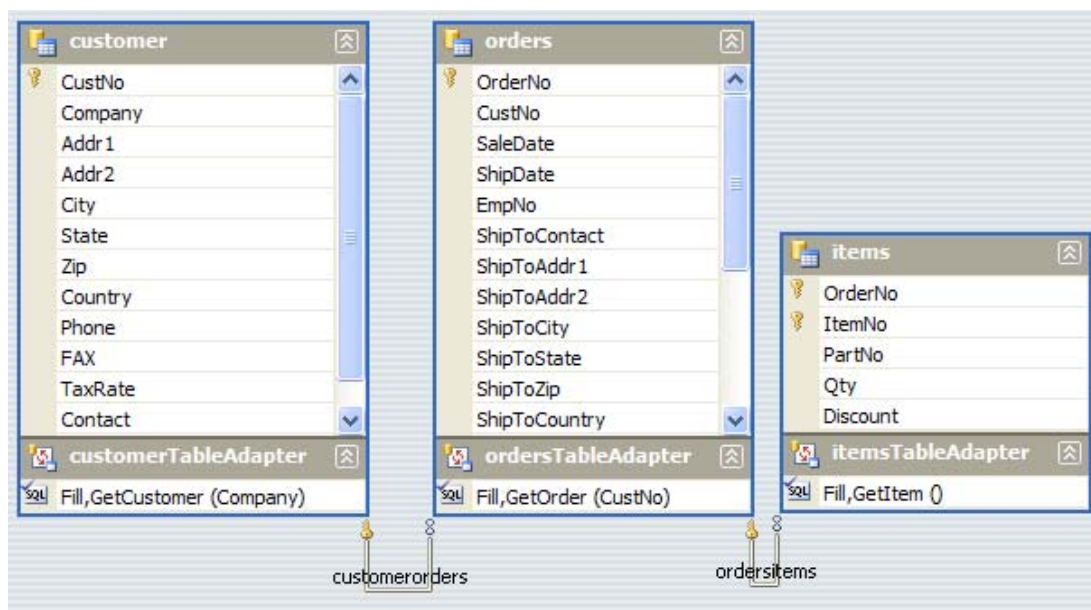
Ici nous simulerons à l'ouverture de la fiche une option de recherche sur le contenu du textbox que nous initialisons à « A ». Dans certains contextes applicatifs il serait intéressant de mémoriser la dernière recherche et de la proposer par défaut à l'ouverture de la fiche ou de l'application, ce n'est pas le cas ici.

Nous savons que ADO.NET propose divers moyens de trier et filtrer les données. Mais nous n'allons pas les utiliser, en tout cas pas le filtre. Pourquoi ? Car le filtre s'applique sur les données en mémoire, déjà chargée, et nous voulons justement limiter le chargement des données.

C'est donc dans la requête SQL du SELECT du DataAdapter de la table customer que nous devons ajouter quelque chose et non nous contenter d'un filtre en mémoire.

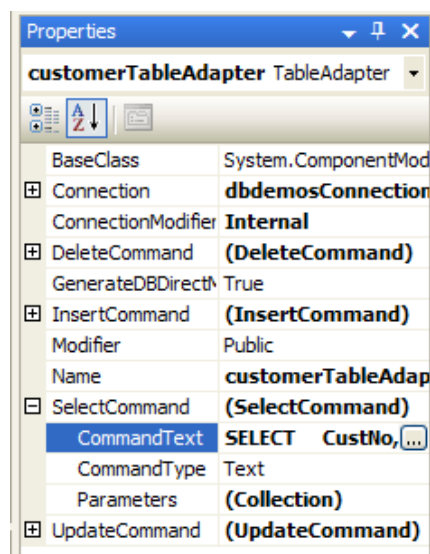
A l'origine l'ordre SELECT avait été construit automatiquement par le concepteur de DataSet typé de Visual Studio. Nous souhaitons maintenant à la fois qu'un tri par ordre des noms de société soit fait par défaut et qu'un paramètre soit ajouté à la requête pour filtrer et limiter les fiches clients à charger. Pour ce faire nous allons tout simplement appeler le schéma du DataSet dans le concepteur en double cliquant sur le fichier DbDemos. xsd.

Le schéma du DataSet est alors affiché :

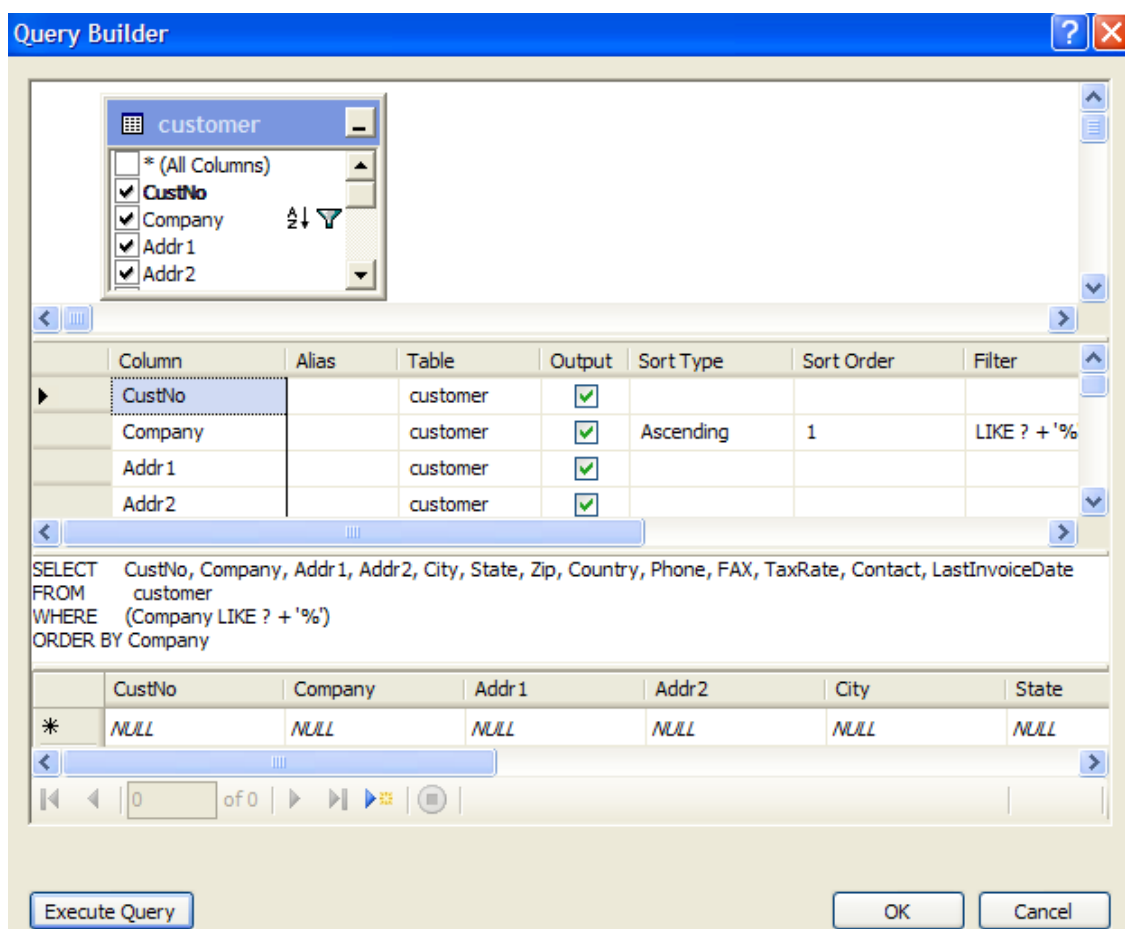


Chaque table y est représentée dans un rectangle. A y regarder de plus près on s'aperçoit que l'espace est découpée en deux zones distinctes : la principale contient les champs de la table, la seconde, en bas, contient les méthodes du DataAdapter spécialisé correspondant à la table considérée.

En cliquant sur l'entête de ce dernier (donc sur customerTableAdapter) l'inspecteur d'objet affichera le détail de cette classe :



Ce qui nous intéresse ici c'est le contenu de la propriété SelectCommand, et principalement la sous-propriété CommandText. En cliquant sur l'ellipse nous affichons le dialogue de saisie de la requête (le query builder) :



Au lieu d'un simple « SELECT * » nous choisissons de cocher tous les champs de la table, le SELECT sera ainsi construit avec la liste nominative des champs.

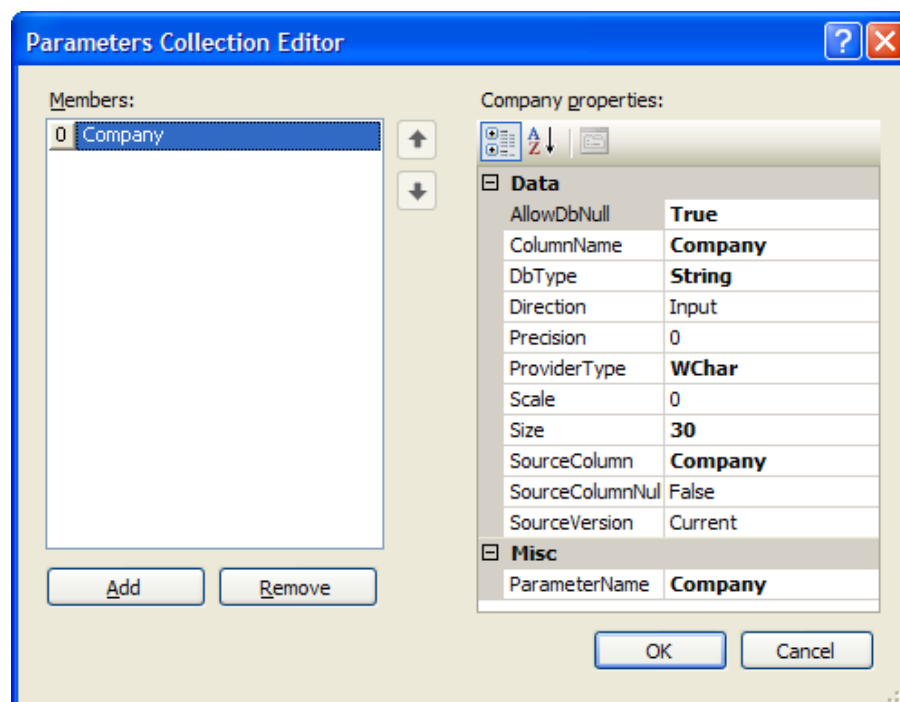
Dans un second temps nous plaçons dans la grille des champs un tri sur la colonne Company (Sort Type : Ascending).

Ensuite nous ajoutons le critère de filtre : « LIKE ? + '%' »

Ce filtre indique que nous comparerons le champ Company à la valeur du paramètre (le point d'interrogation) auquel nous ajoutons le signe pourcent. Il s'agit là d'une syntaxe SQL classique permettant de sélectionner tous les enregistrements dont le champ Company débute par le paramètre (une fois celui-ci renseigné).

Visual Studio étant fort bien fait, le point d'interrogation sera transformé en un objet paramètre dans la DbCommand de sélection. Pour s'en convaincre après avoir validé le dialogue du Query Builder et nous cliquerons sur la propriété Parameters de la commande SELECT du DataAdapter. Avant cela, à la fermeture du dialogue du Query Builder, VS demande si nous souhaitons mettre à jour le DataSet. Il faut répondre par l'affirmative. En effet il va modifier la méthode Fill () afin de prendre en compte le paramètre que nous avons introduit.

Nous pouvons dès lors voir le dialogue des paramètres :



On constate que le paramètre Company a été créé. Les valeurs par défaut nous conviennent. Le nom du paramètre est celui du champ filtré, encore un automatisme de VS qui fait gagner du temps.

Grâce à ce travail sur le schéma, Visual Studio a pu modifier le code source de DbDemos.Designer.cs, un code qu'il nous évite d'écrire et qui compte maintenant 4583 lignes...

Le DataSet

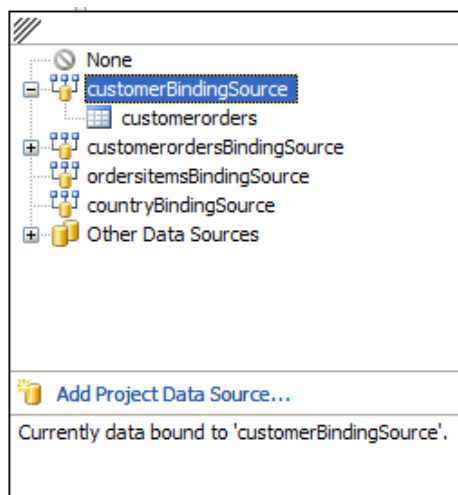
Dans la première partie de cet article nous chargeons les données de toutes les tables en même temps en passant par un DataSet que nousinstancions par code. Il n'est plus nécessaire de procéder de la sorte et nous allons simplement placer un DataSet sur la fiche.

En prenant un DataSet de la palette Données de Visual Studio ce dernier est assez intelligent pour savoir que nous avons défini une classe de DataSet typé dans le projet. Dès lors, au lieu de poser un simple DataSet générique, il affiche un dialogue nous demandant si nous voulons ajouter un tel DataSet ou bien une instance de notre DataSet typé. C'est cette dernière option que nous choisissons.

Maintenant nous disposons d'un composant dbDemos1 placé sur la fiche et il sera facile de connecter les grilles et les autres composants visuels aux tables qu'il contient.

La grille

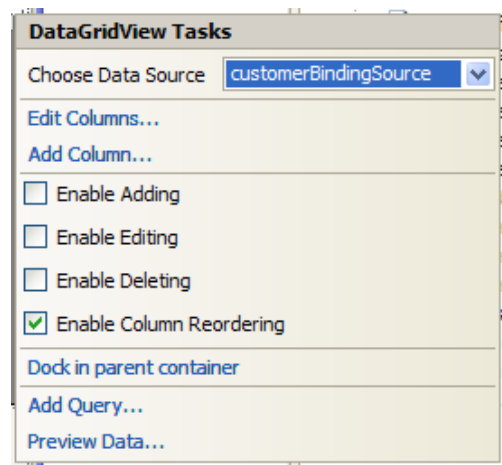
La grille des clients est connectée à la source de données de la table customer dans le DataSet DbDemos1 (plus exactement à la source de binding). Visual Studio placera automatiquement une source de données (customerBindingSource) et un DataAdapter (customerTableAdapter) sur la fiche.



Les autres sources de binding visibles sur la copie écran ci-dessus correspondent aux autres grilles qui ont été connectées.

Nous modifions la grille pour qu'elle affiche les champs Société (Company), Code (CustNo), Ville (City) et Contact (Contact). Comme nous utilisons un tri par défaut sur les noms de société et que cela correspond aussi aux possibilités de recherche, le champ nom de société sera placé en première colonne de la grille.

Nous déconnectons les possibilités d'édition directement dans la grille car nous voulons obliger l'utilisateur à passer par la fiche détail pour ces opérations :



A noter que VS a placé pour nous dans le `Form_Load()` un appel à la méthode `Fill()` du `DataAdapter` pour remplir la table des clients. Nous allons supprimer cette ligne pour la remplacer par notre propre logique.

Notons enfin que la grille est en mode read-only et que la multi-sélection activée par défaut a été déconnectée et que le mode de sélection a été modifié en `FullRowSelect`.

Les champs de la fiche

Sous la grille de données nous plaçons les champs qui simuleront la fiche complète d'un client.

Vous remarquerez que par souci de réalisme le champ pays est édité au travers d'un composant `ComboBox`. Sa source de données est un `DataSet` typé lié à la table `country` de la base de données. Le paramétrage de ce composant n'est pas notre propos et nous supposons ici que cela ne vous posera pas de problème.

Les composants sont liés à la source de données `customerBindingSource`.

Les boutons de commande

Sous la fiche détail du client en cours de sélection nous plaçons deux boutons :

Ces boutons permettent de mettre à jour la fiche du client si elle est modifiée ou d'annuler la modification en cours.

La propriété `Enabled` de ces boutons est placée à `false` dans le concepteur visuel. Les boutons ne seront activés que si les données de la fiche détail du client en cours sont réellement modifiées ce qui est important du point de vue ergonomique.

La philosophie de mise à jour

Arrêtons quelques instants sur la philosophie de mise à jour des clients.

Nous nous plaçons dans le cadre d'une application réelle, donc nous ne voulons surtout pas tomber dans les pièges évoqués en début d'article à propos des verrous et autre vision trop optimiste de leur gestion. Nous ne souhaitons donc pas que l'utilisateur puisse modifier plusieurs fiches clients, que cela reste en mémoire, et que la base de données soit mise à jour plus tard, quand l'utilisateur sortira de l'application, de la fiche, ou qu'il cliquera sur un bouton de mise à jour. Trop de temps se sera écoulé et les risques de conflits seront alors très importants.

Bref, pour sortir de l'alternative verrou pessimiste ou stratégie optimisme risquée, nous avons choisi une troisième voie : la mise à jour client par client avec acceptation ou rejet des modifications pour chacun. Cela limite les risques de conflit à une seule fiche client.

De fait les deux boutons placés sous la fiche détail du client en cours de sélection vont servir cet objectif : dès qu'un élément de cette fiche est modifié les deux boutons deviennent actifs et la grille est désactivée pour que l'utilisateur ne puisse plus naviguer et changer de client. Sa seule option : cliquer sur l'un des boutons pour valider ou rejeter ses modifications.

Cela n'est pas très compliqué à mettre en œuvre mais plusieurs points sont à prendre en compte :

- la détection d'une modification dans l'un des champs de la fiche détail
- l'activation des boutons et leur désactivation
- le blocage et le déblocage de la grille
- la gestion du conflit lors de la mise à jour

Afin de centraliser la modification de l'état des divers composants nous créons une propriété :

```

private bool _CustomerModified;
/// <summary>
/// Propriété indiquant l'état du record en
/// cours (modifié ou non)
/// </summary>
private bool CustomerModified
{
    get
    {
        return _CustomerModified;
    }
    set
    {
        _CustomerModified = value;
        // On répercute l'état sur les éléments visuels
        btnUpdateCustomer.Enabled = value; // bouton MAJ
        btnCancelCustomerChanges.Enabled = value; // annuler
        CustomersGrid.Enabled = !value; // grille on/off
        btnFindCustomer.Enabled = !value; // recherche
    }
}

```

La propriété `CustomerModified` indique en permanence si la fiche détail a été modifiée et sa méthode `set` s'occupe de changer l'état des composants d'interface. Cette stratégie d'une propriété qui gère l'un des états de l'automate (l'application) et qui centralise les interactions avec l'IHM est à conseiller dans tous vos développements.

Reste à savoir comment la propriété sera mise au courant de la modification de la fiche détail... La version la plus logique serait que la source de binding possède un événement activé quand elle passe en mode édition, mais un tel événement n'existe pas. L'option la plus simple consiste donc à créer un gestionnaire d'événement centralisé qui interceptera les modifications des contrôles de saisie, ce que nous verrons un peu plus bas.

Le code des boutons

L'annulation de la mise à jour se fait de la façon suivante :

```

private void btnCancelCustomerChanges_Click(object sender,
    EventArgs e)
{
    // Annulation du mode d'édition de la binding source
    customerBindingSource.CancelEdit();
    // Annulation du drapeau
    CustomerModified = false;
}

```

La mise à jour des données est réalisée comme suit :

```

private void UpdateCustomer()
{
    // valide les modifications en cours et les
    // reporte dans la table
    customerBindingSource.EndEdit();
    try
    {
        customerTableAdapter.Update(dbDemos1.customer);
        CustomerModified = false;
    }
    catch (DBConcurrencyException)
    {
        MessageBox.Show("Impossible de mettre à jour la fiche" +
            "(accès concurrents). Les données vont être " +
            "rechargées.");
        int currentCustomer = Convert.ToInt32(
            CustomersGrid.SelectedRows[0].Cells["CustNo"].Value);
        LoadCustomerData();
        int newpos = -1;
        for (int i=0; i<CustomersGrid.RowCount; i++)
        {
            if
                (Convert.ToInt32(
                    CustomersGrid.Rows[i].Cells["CustNo"].Value)
                    == currentCustomer)
                { newpos = i; break; }
        }
        if (newpos > 1) CustomersGrid.CurrentCell =
            CustomersGrid.Rows[newpos].Cells[0];
    }
}

```

Vous remarquerez que les deux séquences (validation et annulation) mettent à false de la propriété CustomerModified. Cela a pour conséquence de remettre l'interface dans son état « neutre » dans lequel, notamment, l'utilisateur peut naviguer dans la grille et peut lancer une recherche.

La séquence de mise à jour semble complexe... Mais rappelez-vous que nous nous plaçons dans un cas réel, non dans un exemple édulcoré et théorique. De fait, dans la vie réelle il faut bien gérer les exceptions. Ici, et bien que nous n'utilisions pas de transaction puisque nous travaillons sur un seul enregistrement d'une seule table, il peut tout de même arriver qu'un conflit soit détecté. Il suffit pour ça qu'un second utilisateur modifie (et valide) la même fiche client juste avant nous.

Dans un tel cas nous devons avertir l'utilisateur. C'est pourquoi nous gérons l'exception DBConcurrencyException. La première chose est d'afficher un message d'erreur clair à l'utilisateur. Ici nous lui indiquons que la mise à jour est impossible en raison d'un accès concurrentiel. Mais que faire ensuite ?

Il est impossible de « corriger l'erreur » par programmation, l'application ne peut en effet prendre aucune décision. Il faut donc se résoudre à perdre les modifications saisies par l'utilisateur.

Mais afin de rendre la chose moins désagréable pour l'utilisateur nous allons :

- noter le code du client qui était en cours de modification
- recharger les données
- re-positionner la grille et la fiche détail sur le même client

Le code un peu long à l'intérieur du catch sert à réaliser ces tâches.

A noter, les exceptions autres que les accès concurrents ne sont pas gérées.

Le chargement des données

Pour charger les données nous créons une méthode :

```
private bool _LoadingCustomer;

/// <summary>
/// Charge les clients.
/// Utilise la textbox de recherche pour filtrer les données
/// </summary>
private void LoadCustomerData()
{
    _LoadingCustomer = true;
    try
    {
        // vidage pour éviter violation de contraintes
        dbDemos1.items.Clear();
        dbDemos1.orders.Clear();
        // la commande SELECT possède un paramètre
        // qui filtre la liste pour limiter le nombre
        // de records remontés en mémoire.
        customerTableAdapter.Fill(dbDemos1.customer,
            tbSearchCompany.Text.Trim());
        // affichage du nombre de clients remontés
        int i = dbDemos1.customer.Count;
        lCount.Text = i + " client" + (i > 1 ? "s" : "");
        // RAZ du flag indiquant si le record en cours
        // est modifié ou non
        CustomerModified = false;
    } finally { _LoadingCustomer = false; }
}
```

La méthode Fill() du DataAdapter a été modifiée par l'expert de Visual Studio pour intégrer le paramètre que nous avons ajouté au SELECT.

La valeur de ce paramètre est obtenue par la propriété Text du textbox de recherche (à noter le Trim() pour supprimer les éventuels espaces, les utilisateurs ne comprennent pas qu'un espace est différent de « rien » pour un ordinateur).

On notera le vidage préalable des tables orders et items. En effet, si des commandes sont chargées et que nous rechargeons un autre jeu de clients (suite à une recherche), les commandes en mémoire n'auront plus forcément de client correspondant... Comme le DataSet gère les relations il s'en suivra

une violation de contrainte sur la clé étrangère « CustNo » (code client) stockée dans orders.

Le même raisonnement s'applique à items vis-à-vis de orders.

L'ordre de vidage des tables est important, il doit être l'inverse du sens de la descente des clés (c'est-à-dire de la cascade imaginaire qui fait « tomber » les clés de la table de plus haut rang dans les relations. Ici, la clé de customer, CustNo, descend dans orders et la clé de cette table descend ou tombe dans items).

Le champ `_LoadingCustomer` est utilisé pour indiquer que le chargement des clients est en cours ou non. Cela est utile pour certains événements qui sont déclenchés durant cette phase et qu'il ne faut pas traiter à ce moment précis mais uniquement une fois le chargement est terminé. Il s'agit là d'une approche particulièrement intéressante et efficace qui peut se généraliser : utiliser des drapeaux indiquant l'état de l'application (chargement, mise à jour d'affichage, de données, calculs longs, etc). Il est plus aisé de tester l'état d'une machine logique si celui-ci est mis à jour là où il le faut que de déboguer ensuite certains effets de bord.

Détection des mises à jour

Nous avons introduit plus haut la propriété `CustomerModified` qui permet de savoir si l'enregistrement en cours est en train d'être modifié. Cette propriété sert aussi à mettre l'interface utilisateur en accord avec l'état actuel, notamment en interdisant la navigation sur d'autres clients ainsi que la recherche tant que la modification n'est pas validée ou annulée.

Mais comment détecter la modification de l'enregistrement en cours ?

Il n'existe pas d'événement simple pour gérer le début de saisie, il faut donc trouver un autre moyen. La solution la plus simple, même si elle semble contraignante, consiste à créer un gestionnaire d'événement centralisé auquel on connectera tous les événements `TextChanged` des textbox et autres éléments d'interface (ComboBox ou autre).

C'est le rôle de la méthode suivante :

```
/// <summary>
/// Gestionnaire de l'événement Changed des controls servant
/// à modifier les informations du client affiché.
/// </summary>
private void CustomerDataChanged(object sender, EventArgs e)
{
    // on indique par le drapeau que le record en
    // cours est modifié
    // ce qui déclenche l'ajustement de l'interface
    // (boutons disabled, etc)
    if (!_LoadingCustomer) CustomerModified = true;
}
```

On notera le test sur le champ `_LoadingCustomer`, de type booléen. Cette variable est placée à true en début de chargement des clients et basculée à false en fin de cette séquence. Cela nous permet de savoir partout où cela est nécessaire si nous sommes en train de charger les données client ou non.

Cela est nécessaire parce que le chargement des clients déclenche par effet de bord certains événements comme celui décrit ici. Ce type d'interférence est parfois sans danger mais il est souvent la cause de bogues assez sournois. Il est donc préférable de toujours gérer des drapeaux pour indiquer certains états particuliers comme peut l'être le chargement des données.

D'ailleurs ici, sans le test du drapeau une erreur d'exécution difficilement compréhensible pour un débutant apparaîtrait (avec les sources de l'application exemple, faites l'essai en supprimant le test...).

A noter : à ce stade la détection des mises à jour n'est pas encore satisfaisante. En effet, puisque nous détectons la modification de la fiche client à partir des événements TextChanged des contrôles de saisie la détection se fera aussi lorsque le contenu des textbox est modifié par un simple changement de client en cours (navigation). Ce cas de figure ne nous intéresse pas et il convient donc d'annuler la détection lorsque l'enregistrement en cours est changé par navigation.

Plusieurs solutions existent, l'une des plus simples consiste à gérer l'événement de changement de position de la source de binding :

```
/// <summary>
/// La position dans la binding source des clients a changé
/// </summary>
private void customerBindingSource_PositionChanged(object
    sender, EventArgs e)
{
    // on annule le drapeau de modification levé à tort
    // par le changement de
    // ligne qui a entraîné le changement de contenu
    // des textbox, donc la levée
    // du drapeau à true.
    if (!_LoadingCustomer) CustomerModified = false;
}
```

La recherche des clients

Le bouton de recherche ne fait rien d'autre que d'appeler la méthode chargement des données puisque la prise en compte du contenu du textbox de recherche y est intégrée.

Un peu d'ergonomie

Pour terminer sur la partie client, ajoutons un zeste d'ergonomie à notre application.

Dans notre application la sélection d'un client peut servir à voir sa fiche et à la modifier mais aussi à accéder à la liste de ces commandes. L'accès à cette liste est simple puisque l'utilisateur doit juste cliquer sur l'onglet « commandes », mais on peut aller un cran plus loin et rendre le logiciel encore plus agréable à moindre coût.

Il suffit pour cela de réagir au double-clic sur la liste des clients.

```
/// <summary>
/// Double clic grille des clients.
/// Ouvre la page des commandes.
private void CustomersGrid_CellDoubleClick(object sender,
    DataGridViewCellEventArgs e)
{
    tabControl1.SelectedIndex = 1;
}
```

Le code est très simple, sur le double-clic nous affectons au TabControl le numéro de la page des commandes (la page zéro est celle des clients, la 1 celle des commandes, la 2 celle des lignes de commande).

Dans le cadre des modifications ergonomiques nous avons un peu travaillé l'aspect de la grille des clients et avons commencé la mise en page de la fiche des commandes. Vous pourrez regarder cela de plus près en consultant directement le code de l'application fourni avec l'article (projet Real Life2).

La page des commandes

Nous n'allons pas ici recommencer le même cycle d'explications détaillées car finalement ce qui a été dit pour la page de clients, sa mise à jour et son lien avec la page des commandes est tout aussi vrai pour la page des commandes, sa mise à jour et son lien avec la page des lignes de commandes.

Toutefois pour boucler la boucle il nous faut aller au moins jusqu'à l'affichage de la page des commandes.

Nous avons vu que l'utilisateur pouvait accéder à cette page en double-cliquant sur un client dans la liste des clients. Il peut aussi y accéder en cliquant sur l'onglet « commandes ».

Dans tous les cas il nous faut détecter ce changement de page car dans notre application « real life » nous n'avons pas pré-chargé toutes les commandes, il faut donc charger celles du client en cours. L'affichage de la page des commandes est le bon moment pour le faire, nous avons réussi à le retarder le plus possible mais il est impossible de différer plus encore cette action.

La détection du changement de page

L'objet TabControl possède un événement déclenché à chaque changement de page, c'est lui que nous utilisons :

```

/// <summary>
/// Changement de page du tabcontrol .
/// Chargement des données selon la page.
private void tabControl1_Selected(object sender,
    TabControlEventArgs e)
{
    if (e.TabPageIndex == 1) LoadOrdersData();
}

```

Nous utilisons l'objet argument pour connaître le numéro d'index de la page qui va être affichée. S'il s'agit de la page 1 (la liste des commandes) nous appelons la méthode de chargement des commandes.

Le chargement des commandes

Le chargement des commandes impose la même modification du DataAdapter que celui des clients, à savoir l'ajout d'un paramètre dans l'ordre SELECT.

Pour les clients nous avons ajouté un paramètre utilisé pour filtrer les noms de société, pour les commandes nous utiliserons un paramètre permettant de filtrer sur le code client (clé étrangère dans la table des commandes, clé primaire de celle des clients).

Nous ne détaillerons pas la manœuvre, nous l'avons fait pour les clients et cela est rigoureusement identique. L'ajout dans le SELECT consiste à placer une clause « WHERE CustNo = ? ». Le point d'interrogation sera transformé ici aussi en objet paramètre de la commande SELECT. Visual Studio modifiera en conséquence la méthode `Fill()` qui prendra dès lors deux paramètres, le premier est l'objet DataTable à remplir, le second sera le code client.

La méthode de chargement des commandes est ainsi :

```

/// <summary>
/// Charge les commandes du client en cours.
/// </summary>
private void LoadOrdersData()
{
    ordersTableAdapter.Fill(dbDemos1.orders,
        Convert.ToInt32(CustNo.Text));
}

```

La valeur du paramètre (numéro de client) peut être extraite de plusieurs façons de la fiche client en cours de sélection. Lors des améliorations ergonomiques de l'application nous avons ajouté un label lié au champ CustNo juste au dessus de la fiche détail du client en cours. Plutôt que d'aller chercher midi à quatorze heures nous extrayons le code client en cours en inspectant la propriété Text de ce label.

Cette façon de faire, très directe et mélangeant code fonctionnel et interface utilisateur n'est pas très propre. Nous nous en contenterons pour cette application exemple, mais dans la réalité nous devrions créer une séparation forte entre IHM et fonctionnel.

Cela serait effectué en créant une propriété « CurrentCustNo » dans notre fiche. A chaque fois que nous aurions besoin de connaître le code du client en

cours nous utiliserions cette propriété. C'est uniquement dans le get de cette dernière que nous pourrions éventuellement utiliser l'astuce du label.

Certains penseront qu'un tel artifice consiste juste à « cacher la poussière sous le tapis ». Ce n'est pas forcément totalement faux, comme il n'est pas faux de dire que la bouteille est à moitié vide, ou à moitié pleine. C'est une question de point de vue. Si nous nous plaçons du côté de la bouteille à moitié pleine, alors non, il ne s'agit pas tant de cacher un code approximatif que de l'isoler proprement du reste de l'application.

C'est cette isolation que nous visons et non le maquillage du code. De fait toute l'application utilisera une propriété « propre » surtout sans faire de référence à l'IHM.

Le code éventuellement « litigieux » se trouve en un seul endroit, clairement balisé et commenté. Libre à nous de le remplacer ultérieurement par un code différent, cela n'aura pas d'influence sur l'application qui elle n'accède qu'à la propriété.

Cette façon d'isoler et de protéger le code est essentielle à comprendre et à appliquer dans les applications réelles.

L'affichage des commandes

La grille ressemble à cela :

	Num.Cde	Vendu le	Livré le	Réglé par	euros HT
▶	1317	01/02/1995	01/02/1995	Check	7 572,00 €
	1294	04/01/1995	04/01/1995	MC	3 304,85 €
	1217	22/11/1994	22/11/1994	Check	51 730,80 €
	1137	27/11/1993	27/11/1993	Credit	6 785,40 €
	1117	13/04/1993	13/04/1993	Check	6 734,85 €
	1099	16/06/1989	16/06/1989	Credit	859,95 €
	1074	19/04/1989	20/04/1989	MC	2 195,00 €
	1037	26/08/1988	27/08/1988	Credit	2 117,00 €

On notera que par souci d'ergonomie nous avons placé au dessus de la grille un rappel du code client et du nom de la société en cours.

Un champ calculé a été ajouté à la table orders afin d'afficher le montant restant dû (à partir des champs TotalAmount et AmoutPaid, total commande et montant réglé). On ne voit pas ce champ sur la capture, il est plus à droite dans la grille mais vous pourrez inspecter le code de l'application et notamment celui du DataSet typé pour voir comment nous avons défini ce champ.

Conclusion

Il s'agissait d'écrire un article, pas un livre... et 37 pages plus loin en corps 11 il faut constater que cela est un peu long pour un simple article !

Mais le sujet est essentiel et cet aspect « réaliste » des choses n'est traité dans aucun livre que l'auteur a pu consulter, qu'il s'agisse de la presse française comme anglo-saxonne.

Passer de la théorie où tout semble clair et rectiligne à la pratique n'a rien d'évident.

Ici l'auteur n'avait nullement la prétention d'écrire quelque chose d'aussi long ni de faire œuvre d'académisme. Il a conscience que la longueur du sujet l'a obligé à écrire ces pages en plusieurs fois, étalées sur plusieurs jours, et qu'il manque de la cohérence à l'ensemble.

Mais ce n'est qu'un article diffusé gracieusement, peut-être cela donnera-t-il naissance à plusieurs chapitres repensés dans un prochain ouvrage...

Dans tous les cas l'auteur espère ici vous avoir permis de mettre le doigt sur des aspects peu traités de la programmation sous ADO.NET.

Puisse cet humble objectif avoir été atteint.

Bon Développement !

O.Dahan