# DotNet : COM interop

**J-M Rabilloud**

**www.developpez.com. Reproduction, copie, publication sur un autre site Web interdite sans l'autorisation de l'auteur.**

## Remerciements

J'adresse ici tous mes remerciements à l'équipe de rédaction de "developpez.com" et tout particulièrement à Etienne Bar, Maxence Hubiche, David Pedehourcq et Romain Puyfoulhoux pour le temps qu'ils ont bien voulu passer à la correction et à l'amélioration de cet article.

Dans cet article nous allons regarder comment communiquer avec les composants COM. Nous allons commencer par voir ce qu'est COM, avant de voir sur quoi repose l'interop COM. Enfin nous verrons un exemple d'utilisation de composant COM dans une application managée.

Bonne lecture.

# Interop COM

Dans les langages DotNet, il est possible de continuer d'utiliser des composants COM sans avoir à réécrire ceux-ci. De même, il est aussi possible d'utiliser des composants managés dans un client COM. L'ensemble de ces mécanismes est défini par le terme interop COM. Pour mieux comprendre en quoi cela consiste nous allons étudier le fonctionnement de celui-ci et commencer par quelques rappels des fondamentaux.

## *Interactions des composants binaires*

Tout cet article va au fond aborder un seul sujet, la réutilisation du code. Le but du standard COM n'est que de permettre l'utilisation de composants déjà écrits sans se soucier de leur réalité physique.

### Component Object Model (COM)

Dans la suite de cet article je vais employer trois termes qu'il ne faut pas confondre :

Une classe est une définition d'un objet. Le terme de classe s'entend de façon conceptuelle.

Un objet est l'instance d'une classe à l'exécution.

Un composant est une entité exécutable composée d'un ou plusieurs objets. Un composant est généralement issu d'un seul fichier contenant toutes les classes définissant ses objets.

COM est un ensemble de règles définissant un standard de communication. Les règles COM permettent d'utiliser un composant sans connaître la réalité physique de son fonctionnement.

Imaginons le composant x. Il possède des propriétés, des méthodes, des données qui lui sont propres ainsi que des règles de fonctionnement interne. En soi, c'est une entité logicielle indépendante. Pour pouvoir être réutilisé, il doit pouvoir communiquer avec d'autres entités logicielles. Pour cela un composant expose une ou plusieurs interfaces. Une interface est en fait le moyen pour un composant d'appeler des fonctions d'un autre composant. On dit couramment qu'une interface est le prototype du composant. C'est donc cette interface sur laquelle vont porter les règles COM.

Une interface doit :

- Etre unique dans son identification
- Etre immuable
- Dériver de l'interface IUnknown

## ***Immuabilité***

C'est le concept le plus facile à intégrer. Du moment qu'un composant peut être amené à évoluer, et à être utilisé par des applications tierces, il faut que l'application n'ait pas à être modifiée si le composant est modifié. Si on fait évoluer le composant, on peut lui ajouter une nouvelle interface donnant accès à de nouvelles fonctions, mais les interfaces précédentes doivent être conservées et les fonctionnalités liées à celle-ci toujours valides.

Si cette règle est facile à concevoir, elle n'en est pas pour autant toujours appliquée, ce qui fait que de nombreux composants soi-disant COM ne le sont pas.

## ***Unicité***

Une interface doit pouvoir être identifiée de façon unique dans l'espace et dans le temps. Sans rentrer dans le détail, disons que chaque interface est identifiée par un Interface IDentifier (IID) celui-ci étant en fait un Globally Unique IDentifier (GUID). Désolé pour ce passage un peu 'jargonnesque' mais cela revient à dire que toutes les interfaces de tous les composants COM ont un identifiant unique. Ceux-ci sont généralement présents dans la base de registre du poste les utilisant, c'est pour cela que l'on dit qu'il faut 'enregistrer' les composants.

## ***L'interface IUnknow***

L'interface IUnknown définit deux règles :

- La navigation entre les interfaces
- La durée de vie du composant

La navigation doit être comprise dans le sens de la possibilité d'obtenir une référence sur n'importe quelle interface d'un même objet.

La durée de vie est un concept un peu plus sioux. Si la classe n'a pas de réalité à l'exécution, ces instances (ou objets) ont eux une réalité temporelle. Un composant existe dans le sens de l'exécution à partir du moment où il existe une référence à l'un de ces objets jusqu'au moment où toutes ces instances ont été détruites. De fait, l'interface IUnknown expose trois méthodes.

QueryInterface ⇒ Permet la navigation entre les interfaces de l'objet.

AddRef ⇒ Ajoute une référence à l'objet

Release ⇒ Supprime une référence à l'objet

Donc un objet gère un compteur interne qui augmente à chaque appel de AddRef et diminue à chaque appel de Release, lorsqu'il vaut 0, l'exécution s'arrête. C'est en cela qu'on dit que le composant gère sa durée de vie.

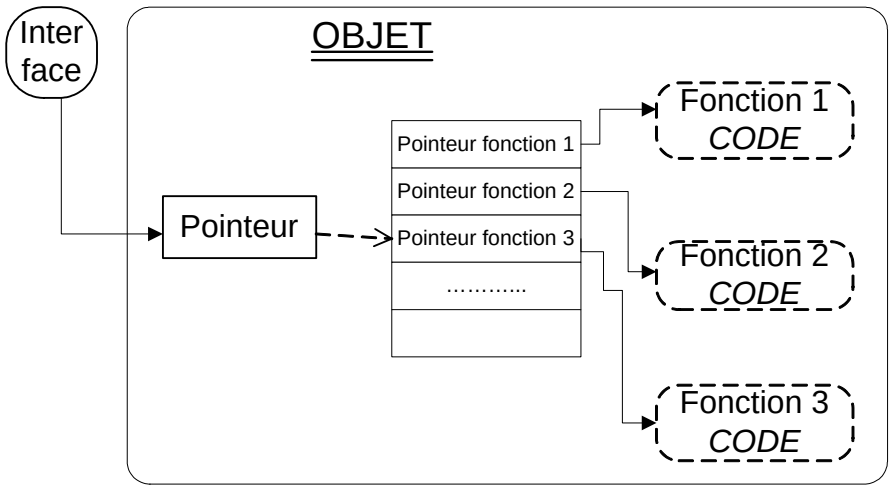
## ***Concepts généraux***

Récapitulons, ce qu'on appelle un composant COM est en fait un ensemble de classe COM définissant des interfaces COM.

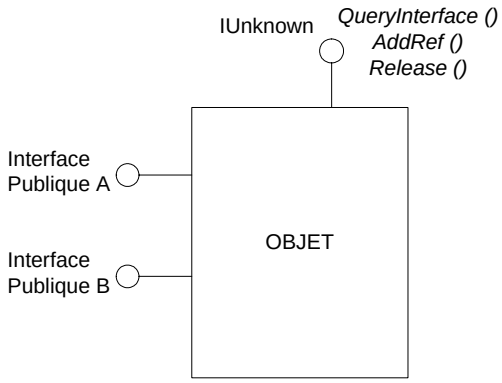
In fine, le composant x gère des fonctions que je peux appeler par le biais d'une de ces interfaces.

Toutefois une interface n'est pas une classe ni un objet. Une interface n'est rien d'autre que le point d'entrée / sortie d'un objet.

Les schémas suivants vont nous permettre de voir la représentation de ce concept.




L'objet expose une interface qui est en fait un pointeur vers un tableau de pointeur de fonction




Les objets COM sont souvent représentés avec ce style de diagramme. On accède généralement à l'interface A ou B par l'appel de QueryInterface de IUnknown qui est toujours exposée.

Tout cela peut paraître assez abscons si vous arrivez de VB6, mais vous l'avez probablement utilisé sans le savoir.

Envisageons le code suivant :

```
Dim MonInst1 As clsClasse, MonInst2 As clsClasse, MonInst3 As clsClasse
Set MonInst1 = New clsClasse
Set MonInst2 = MonInst1
Set MonInst3 = MonInst2
Set MonInst2 = Nothing
Set MonInst1 = MonInst3
```

Lorsque vous écrivez cela, VB6 lui comprend :

```
Dim MonInst1 As CLSID_ clsClasse = 0, MonInst2 As CLSID_ clsClasse = 0,
MonInst3 As CLSID_ clsClasse = 0
MonInst1 = CreateInstance(CLSID_clsClasse, IID_clsClasse)
MonInst2 = IUnknown(MonInst1).QueryInterface(IID_clsClasse)
MonInst3 = IUnknown(MonInst2).QueryInterface(IID_clsClasse)
IUnknown(MonInst2).Release
MonInst2=0
IUnknown(MonInst1).Release
MonInst1 = IUnknown(MonInst3).QueryInterface(IID_clsClasse)
```

On ne voit pas les AddRef car ils sont automatiquement appelés par QueryInterface.

En VB6, toutes les classes implémentent IUnknown de façon transparente. Vous n'avez d'ailleurs pas besoin non plus d'implémenter explicitement IID_clsClasse car VB6 le fait pour vous comme interface par défaut. C'est d'ailleurs un des problèmes de VB6 que de masquer beaucoup de ses actions.

Bon! Personne ne s'est endormi ? Je continue.

## Exécution

Il existe deux types d'exécutions pour les composants.

- Ceux-ci peuvent s'exécuter dans le même processus que l'application cliente, on parle alors de composants in-process. Comme l'espace d'adressage est le même, la communication entre l'application et le composant est directe et le fonctionnement très rapide. Malheureusement une erreur du composant peut entraîner un arrêt de l'application. Ce sont principalement des bibliothèques dynamiques (DLL) parfois appelées OCX quand elles présentent une interface utilisateur graphique.

- Ils peuvent aussi s'exécuter dans leurs propres processus, ce qui est le cas des applications office par exemple, on parle alors de composant out-of-process ou de 'serveur de composants'.

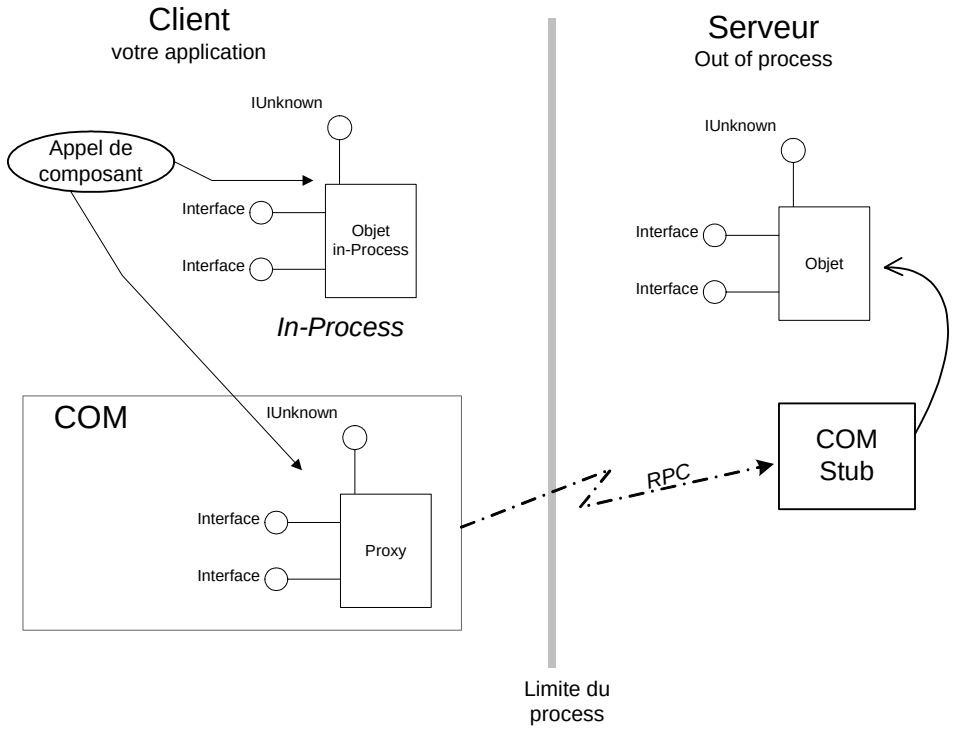
Dans ce cas, on sort du modèle COM strict pour accéder à la notion de COM distribué, notée DCOM. Ce fonctionnement est bien plus complexe donc attachez vos ceintures. Lorsque vous demandez une instance d'un objet, vous communiquez à des API OLE32 l'identificateur de l'objet. Celle-ci localise physiquement le composant, l'exécute et vous renvoi le pointeur d'interface demandé. Comme il n'y a pas de possibilité de communication directe entre deux processus différent, Il va y avoir mise en place du marshalling COM. Notez que là encore je fais simple. Plusieurs mécanismes sont mis en jeu selon que le serveur est local ou distant.

### Marshalling

Il y a chargement dans le processus client, l'application par exemple, d'une DLL appelée le Proxy. Celle-ci va simuler côté client le composant. Dans le composant il y a un objet similaire, nommé le Stub. Entre le Proxy et le Stub il y a communication par un protocole indépendant (RPC). Le client voit donc le Proxy dans son espace d'adressage, celui-ci simule toutes les interfaces du composant.

Lorsque le client veut utiliser une méthode du composant il adresse les paramètres de la méthode appelée à l'interface simulée par le Proxy. Celui-ci les communique au Stub du composant qui simule le client pour le composant. S'il y a des valeurs de retour, le fonctionnement inverse est effectué.

Ce fonctionnement n'a lieu qu'avec des paramètres de méthodes. Il n'est bien sûr possible que parce que le Proxy sait simuler le composant, donc connaît les interfaces. Pour y parvenir, le composant doit fournir son propre Proxy.




### Description des interfaces

Le fonctionnement que je vous ai décrit présente deux problèmes évidents. Le développeur doit connaître le composant pour l'utiliser, et chaque composant doit véhiculer sa propre logique de communication. Pour pallier à ces inconvénients, on utilise un langage de description des interfaces 'Interface Definition Language' (IDL). Ce langage est en fait un standard qui permet de décrire des interfaces compréhensibles par tous.

## *Automation*

Faisons une petite pause. Je suis resté très superficielle dans le fonctionnement et j'ai fait de nombreuses impasses sur des mécanismes utilisés par COM car dans le cas du pilotage Office c'est sensiblement un autre mécanisme que nous allons utiliser.

Jusqu'à présent, le composant décrit doit être référencé dans votre application, et seul un code compilé peut utiliser le composant. Mais le langage IDL permet de décrire une interface particulière, IDispatch.

### Bibliothèque de type (type library)

Les interfaces du composant sont donc décrites dans un langage particulier. Si je compile cette description, j'obtiens une bibliothèque de type. C'est en fait la définition des propriétés et méthodes du composant. Vous allez vous dire que je délire avec mes propriétés de composant puisqu'on ne peut Marshaller que des méthodes. Et vous avez partiellement raison. En fait un composant expose des propriétés par le biais de méthode ; une pour la lecture de la propriété l'autre pour l'écriture. Une propriété en lecture seule est donc une propriété dont l'interface expose la méthode de lecture mais pas celle d'écriture.

Ma bibliothèque de type peut être référencée comme le composant auquel elle fait référence puisqu'elle possède toutes les informations du composant. C'est toujours une bibliothèque de type qu'on utilise pour manipuler les applications Office.

### Composants programmables

Les composants qui implémentent l'interface IDispatch ont une particularité bien plus importante, ils peuvent décrire leurs interfaces à l'exécution. Ce mécanisme est appelé 'Automation'. Cela implique de nombreuses possibilités.

Comme le composant peut se décrire à l'exécution, je ne suis pas obligé de le référencer pour l'utiliser. Ceci lui permet d'être utilisable par les langages non compilés, on parle aussi de composants 'programmables'. Il n'y a pas nécessité alors de lier le composant lors de la compilation, c'est la liaison tardive.

### Liaison tardive

Un composant COM non programmable doit être référencé dans l'application qui l'appelle. Les paramètres des méthodes de ces composants étant typés, il y a vérification lors de la compilation de l'existence de la méthode et du type des paramètres transmis, c'est la liaison précoce. Lorsqu'on travaille en code compilé, on recommande fortement d'utiliser ce type de liaison puisqu'elle est plus rapide est plus fiable, mais si le composant supporte Automation elle n'est pas obligatoire. Comme le composant peut se décrire à l'exécution, il est tout à fait possible d'appeler ses méthodes sans que le compilateur ne fasse aucune vérification.

Cela n'est d'ailleurs pas sans danger. En effet, il arrive fréquemment que l'on référence un composant et qu'on l'utilise dans ce qu'on croit être une liaison précoce. Si un lien ne peut être résolu lors de la compilation et que le composant implémente une interface IDispatch, il n'y aura pas d'erreur de compilation et l'appel sera résolu à l'exécution. On sera alors en liaison tardive sans le savoir.

## Allons plus loin

### *Process, Threads & appartements*

Votre application lorsqu'elle s'exécute forme un process. Pour simplifier disons que cela représente une entité exécutable représentant une pile mémoire et un espace d'adressage. Un process est cloisonné vis-à-vis de l'extérieur.

Un thread est une unité d'exécution. Sur une machine n'ayant qu'un processeur, le concept de multi tâches est une vue de l'esprit. Il s'agit en fait de tâches en temps partagé, c'est à dire que chaque tâche se verra allouer du temps par le processeur. Chaque thread actif représente une tâche.

Un Process peut contenir un ou plusieurs threads. Lorsqu'il en contient plusieurs, il peut soit les gérer selon le mode libre, c'est alors le composant qui gère le multithreading, soit dans le mode dit 'Cloisonné' il n'y a alors pas de multithreading. La gestion des problèmes de synchronisation, d'inter blocage et de réentrance diffère selon ce mode.

Quel que soit le mode, c'est la notion d'appartement que je souhaite vous faire toucher du doigt.

Un appartement peut être composé d'un ou de plusieurs threads connexe. Un objet n'existe que dans un seul appartement. Il doit forcément y passer toute sa vie. Les objets d'un appartement ne peuvent communiquer directement qu'avec les objets du même appartement. Pour toutes les communications entre appartement il y a Marshalling.

### *Fabrique de classe*

Comme nous l'avons vu, un composant COM doit pouvoir instancier ses objets indépendamment de la nature (le langage) de l'appelant. Pour qu'il puisse créer des objets, on utilise une fabrique de classe qui elle, sait instancier les objets du composant et renvoyer une référence sur ces objets. Ces fabriques de classe sont toujours enregistrées dans la base de registre, et sont souvent improprement appelées coclasse.

Les fabriques de classes sont souvent appelées 'objets exposés par le composant'.

### *Délégation & Agrégation*

Il arrive fréquemment qu'un objet COM englobe d'autres objets COM. Cependant il peut y avoir deux cas assez différents.

On dit qu'il y a délégation si l'objet inclus n'expose aucune de ces interfaces à l'extérieur. A ce moment là, c'est l'objet conteneur qui gère les services des objets inclus.

Il y a agrégation si l'objet inclus expose au moins une de ses interfaces vers l'extérieur. Dans ce cas l'interface IUnknown du conteneur peut donner une référence sur l'interface exposée de l'objet contenu.

### *Gestion et Propagation des erreurs*

Dans un monde idéal, le composant gère ses erreurs. Toutefois il doit pouvoir informer le client dans certains cas. Pour communiquer des erreurs, on utilise un Entier long appelé code HRESULT qui contient trois champs (le numéro de l'erreur, la sévérité et le service). Certain langage de programmation comme VB6 ne gère pas de renvoie de valeur HRESULT explicite puisqu'il propage une exception. N'oubliez pas que le composant se doit de gérer au maximum ses erreurs en interne.

### *Conclusion*

Il existe des livres et des articles très complets sur la programmation des composants sur le standard COM. Nous n'avons vu ici que le strict nécessaire à la compréhension de leur fonctionnement.

## .NET Framework

Le Framework est un environnement managé par nature très différent du modèle COM, même s'il y a aussi de nombreuses similitudes.

Commençons par le commencement donc, qu'est ce qu'un environnement managé. Ce terme vient de l'Anglais "managing software risk". Pour schématiser, cela revient à imaginer une couche entre le système et l'application. Un code dit managé est un code qui est écrit pour fonctionner avec cette couche. Dans le cas de DotNet, il s'agit du Common Language RunTime (CLR).

La première de ses fonctions consiste à gérer la sécurité. Lorsqu'on utilise un code compilé, on utilise les propriétés d'un système. Pour être simplement portable entre plusieurs Windows différents, on est obligé de perdre en qualité puisque tous les Windows ne gèrent pas la sécurité à l'identique. Comme le CLR prend en charge la sécurité, le problème ne se pose plus dans ce type d'environnement. Comme cela sort du cadre de cet article, je ne vais pas parler plus de sécurité et nous allons nous concentrer sur les points qui nous intéressent.

## ***Gestion de la mémoire***

Dans les applications de types compilées, chaque composant / application gère sa mémoire.

Chaque process alloue / désalloue sa mémoire comme il l'entend. Ces types de programmes sont dits "à finalisation déterministe". Ceci veut dire que lorsque l'on demande la libération des ressources (de la mémoire) celle-ci à lieu (si tout se passe bien). Dans un environnement managé, la mémoire est gérée par l'environnement. Dans notre cas, c'est le CLR qui gère la mémoire.

Le modèle classique est simple. Comme dans un environnement compilé il existe des process et des threads. Un thread correspond à une unité d'exécution auquel sera alloué du temps processeur. La mémoire sera gérée par un gestionnaire centralisé, appelé "Garbage Collector". Ce processus est appelé aussi gestion automatique de la mémoire.

Cette technique est généralement non déterministe. Lorsque vous libérez des ressources, celles-ci ne sont effectivement libérées qu'au prochain passage du Garbage Collector. Je dis généralement car les composants managés peuvent exposer une interface IDispose qui rend leur finalisation déterministe.

## ***Méta données & types***

Les classes DotNet utilisent des méta-données pour leurs descriptions. Cela revient à dire qu'il n'y a plus besoin d'un langage de définition des interfaces. Un composant managé peut toujours se décrire lui-même à l'exécution. En cela c'est assez différent du modèle COM standard. Ce processus d'auto information est appelé réflexion. Une bibliothèque de type COM donne la hiérarchie des objets et la définition des interfaces dans une bibliothèque de type. Un composant managé donne le même modèle en exposant ses méta-données au travers de la réflexion.

Le CLR ne gère pas non plus les types comme COM. Comme nous le verrons plus loin, il faut donner les règles de conversion entre les types COM et les types managés.

## ***Composants Managés***

Les composants managés évoluent dans un environnement très différent de leurs homologues COM.

- Un composant managé ne gère pas sa durée de vie
- Un composant managé fournit ses services par réflexion
- Un composant managé peut être déplacé par le Runtime
- Un composant managé ne nécessite pas d'inscription dans le registre
- Enfin le composant managé s'exécute dans la mémoire managée.

Cela fait quand même de nombreuses différences, et il y en a encore d'autres. Pourtant il est possible d'utiliser des composants COM avec le Framework en utilisant le même schéma fonctionnel que pour les composants out of process. C'est ce mécanisme qu'on appelle l'interop COM.

## *Principe de fonctionnement*

La barrière environnement managé / non managé va s'envisager comme la barrière inter process. Ceci n'est en rien complexe. Nous allons voir en plus que cela fonctionne dans les deux sens puisqu'il est aussi possible d'exposer des composants managés à des applications non managées.

### Primary Interop Assembly (PIA)

Le premier exercice va toujours consister à créer un assembly ou un code managé permettant la conversion des types COM vers les types DotNet, ou autrement dit, à générer les méta-données managées d'un composant COM. On peut distinguer deux cas.

#### *Existence d'une bibliothèque de type COM*

Une bibliothèque de type n'est pas toujours sous la forme d'un fichier TLB. Celle-ci peut être incorporée dans la DLL (ou l'OCX).

Pour obtenir l'assembly il y a alors plusieurs choix possibles. Les exemples suivants vont être faits sur SQLDMO qui permet la manipulation des structures de données SQL-Server.

##### PIA 'fabricant'

C'est forcément le cas idéal. Dans certains cas, un assembly est déjà disponible pour le composant. C'est par exemple le cas pour Office 2002 dont les PIA sont disponibles chez Microsoft

##### Utilisation de VS.NET

C'est le cas général. La construction de l'assembly se fait par l'intermédiaire de l'IDE.
Sélectionnez le menu "Projet" – "Ajouter une référence"
Sélectionnez l'onglet COM
Vous voyez la liste des composants que l'IDE peut convertir. Il faut pour cela que le composant soit enregistré. Sinon vous pouvez peut être l'atteindre en cliquant sur le bouton parcourir.
Sélectionnez le composant puis cliquez sur "Sélectionner" puis sur "OK"
Si tout se passe bien, vous voyez apparaître dans les références de l'explorateur de solutions une DLL qui correspond à votre PIA (Dans mon cas interop.SQLDMO.dll)

##### Utilitaire Type Library Importer

Il s'agit d'un utilitaire en ligne de commande. Comme je ne suis pas payé à la ligne (ni au reste d'ailleurs), je vous invite à lire la rubrique correspondante dans l'aide MSDN pour voir la liste des commutateurs. Le programme (TlbImp.exe) se trouve normalement dans le répertoire Visual Studio sous le chemin SDK\v1.1\bin. Pour plus de renseignement regardez dans MSDN à « Type Library Importer (Tlbimp.exe) » et « Type Library Exporter (Tlbexp.exe)

##### Classe TypeLibConverter

Cela permet d'obtenir un résultat similaire aux précédents mais de façon dynamique (à l'exécution). Bien qu'on utilise rarement cette possibilité, elle permet de travailler uniquement en mémoire.

#### *Bibliothèque de types inaccessible*

Dans ce cas, il faut générer soit même l'Assembly. Bien que ce ne soit pas nécessairement très compliqué, c'est assez fastidieux. En plus cela demande une bonne connaissance des co-classes du composant. Mais avant tout cela demande de bien comprendre ce qu'il se passe lors des appels à l'interop COM.

## Les Wrappers

Pour qu'il y ait possibilité de communication .NET / COM, il faut donc un mécanisme pour franchir la barrière de l'environnement managé. DotNet expose pour cela des classes particulières nommées Wrappers. Il en existe deux types :

- celles permettant d'utiliser un composant COM dans une application managée, c'est à dire par le runtime sont appelées 'Runtime callable wrapper' (RCW)
- celles permettant d'utiliser un composant managé dans une application native, c'est à dire par COM sont appelées 'COM callable wrapper' (CCW)

Ces Wrappers vont fonctionner de façon assez similaire au Proxy et au Stub de COM.

Récapitulons ici les différences.

Un composant gère sa durée de vie par comptage de ses références, la durée de vie d'un composant managé est gérée par le Runtime.

Les méthodes d'un composant COM ne sont accessibles que par des interfaces, un composant managé les expose par réflexion.

Les Wrappers, quel que soit leur sens vont donc devoir simuler le client dans le serveur et inversement

### *Fonctionnement des Wrappers*

Lorsque dans le code client on fait appel à un objet COM, le Runtime va donc créer un RCW dont le but va être de gérer les références à l'objet COM. Il y a donc création d'autant de Wrapper que d'objet. Le Wrapper va alors exposer les méthodes de l'objet COM au code managé. Chaque appel vers une des méthodes de l'objet COM passe donc par le RCW qui le traduit par un appel sur l'interface correspondante de l'objet COM. Le RCW expose donc toutes les interfaces de l'objet au code client et gère toutes les références du composant. On peut dire alors qu'il existe deux visions du Wrapper. Lorsque le client .NET libère l'objet, le RCW libère la référence du composant, celui-ci devant gérer son cycle de vie normalement. Le RCW peut alors être collecté.

Les types sont transformés si besoin est par le RCW. C'est au niveau du Wrapper que le PIA va être utilisé.

Pour pouvoir faire transiter les paramètres, il va falloir procéder de la même façon que dans DCOM, c'est le Marshalling Interop.

### *Marshalling Interop.*

Le runtime fournit des classes, appelées wrapper qui vont simuler le composant comme s'il était dans l'environnement du client, tout comme le font le Proxy et le Stub. Ce mécanisme s'appelle le Marshalling Interop.

Comme le marshalling COM, le marshalling Interop est un mécanisme intrinsèquement dynamique. Il a donc un coût en terme de vitesse d'exécution.

Globalement, le marshalling interop consiste à échanger des données entre la pile managée et celle du composant. Il y a toutefois deux erreurs à ne pas commettre.

- Il peut y avoir simultanément marshalling COM et marshalling interop pour peu que les objets COM ne soient pas dans le même appartement.
- Il y a toujours intervention de DCOM pour les composants distants

Les règles du Marshalling interop sont les mêmes que celles du marshalling COM.

Le CLR prend en charge la désallocation de la mémoire. Il y a utilisation de paramètre d'entrée (ByVal) ou d'entrée / sortie (ByRef).

Le problème repose principalement sur la conversion des types. Lorsqu'un type a une représentation dans la mémoire managée et non managée, il est dit 'blittable'. C'est par exemple le cas des entiers.

Sinon il y a conversion de type. Les règles détaillées se trouvent dans la documentation, mais nous allons regarder deux cas.

### Chaîne de caractère

COM utilise des chaînes de type BSTR (tableau de caractère terminé par le caractère Null) et des caractères ANSI. Une chaîne managée est une collection séquentielle de caractère unicode. Normalement pour les chaînes de longueur variable, généralement utilisée comme paramètre de méthode, il n'y a pas de difficulté. Par contre il n'est pas possible de passer une chaîne de longueur fixe. On doit alors forcément utiliser un objet StringBuilder.

### Variant

Il n'y a pas de Variant dans le CLR. Dans le CLR, le type universel est 'Object'. Il y a donc toujours conversion de type des variants entrant. Lorsqu'un variant est passé de COM vers le code managé, il y a lecture du sous-type du variant lors du marshalling pour la conversion. Du fait que variant comme object sont des types 'universels', on ne peut jamais être sur qu'un variant aura le même sous-type lorsqu'il sera retourné à l'objet COM.

## Conclusion

Nous avons maintenant les connaissances de bases nécessaires pour faire de l'Interop COM, ce que nous allons faire maintenant dans l'exemple suivant.

# Exemple d'utilisation

## *Créer le composant StoreProc.dll*

Dans cet exemple, je vais créer un composant simple. Celui-ci utilise lui-même une autre DLL, SQL-DMO. Celle-ci permet la manipulation de structure SQL-Server.

Mon composant va être extrêmement simple. Le but est de pouvoir lire le code d'une procédure stockée, ce que l'on ne peut pas faire aisément avec MSDE par exemple.

Je vais créer ce composant avec VB6 pour garder une certaine homogénéité d'écriture avec VB.NET. Toutefois comme nous l'avons vu, le langage de création ne change rien.

Je crée donc une DLL. Je définis tout d'abord la propriété 'Instancing' à MultiUse, ce qui veut dire que n'importe quelle application peut créer des objets à partir de ma classe (Fabrique de classe). Je la définis aussi comme non persistante, c'est à dire qu'une propriété ne peut pas être conservée entre deux instances. Enfin dans les propriétés du projet je choisis compatibilité binaire afin que le CLSID de toutes les versions soit maintenu. Cela permet aux programmes utilisant une ancienne version de ma classe de fonctionner.

J'ajoute aussi une référence à SQLDMO.DLL dans mon projet. Voici le code de ma classe

```
'variables locales de stockage des valeurs de propriétés
Private mvarEstConnecté As Boolean 'copie locale
'Instances d'objets de SQLDMO
Private MonServeur As SQLDMO.SQLServer
Private MaBase As SQLDMO.Database

Public Property Get EstConnecté() As Boolean
    mvarEstConnecté =
MonServeur.VerifyConnection(SQLDMOConn_CurrentState)
    EstConnecté = mvarEstConnecté
End Property

Public Sub Deconnexion()

    MonServeur.DisConnect

End Sub

Public Function ChaineProc(ByVal NomProc As String) As String

    On Error Resume Next
    ChaineProc = MaBase.StoredProcedures(NomProc).Text
    If Err.Number > 0 Then
        MsgBox "la procédure " & NomProc & " n'existe pas",
vbCritical, "ERREUR"
    End If
End Function

Public Function ListeProc() As Variant
    Dim Procedure As SQLDMO.StoredProcedure, NbProcedure As Integer,
cmpt As Long, TabChaine() As String

    NbProcedure = MaBase.StoredProcedures.Count
    If NbProcedure > 0 Then
        ReDim TabChaine(0 To NbProcedure - 1)
        For cmpt = 0 To NbProcedure - 1
            TabChaine(cmpt) = MaBase.StoredProcedures(cmpt + 1).Name
        Next cmpt
        ListeProc = TabChaine
    End If

End Function
```

```
Public Function Connexion(ByVal Serveur As String, ByVal Catalogue
As String, ByVal User As String, ByVal Password As String) As
Boolean

    On Error GoTo Echec_Connect
    Set MonServeur = New SQLDMO.SQLServer
    MonServeur.LoginSecure = True
    MonServeur.Connect Serveur, User, Password
    Set MaBase = New SQLDMO.Database
    Set MaBase = MonServeur.Databases(Catalogue)
    mvarEstConnecté = True
    Connexion = True
    Exit Function
Echec_Connect:
    Err.Clear
    Connexion = False
    On Error GoTo 0

End Function

Private Sub Class_Terminate()

    If MonServeur.VerifyConnection(SQLDMOConn_CurrentState) Then
MonServeur.DisConnect
    Set MaBase = Nothing
    Set MonServeur = Nothing

End Sub
```

Comme vous le voyez, elle possède une propriété en lecture seule, puisqu'il n'existe pas de fonction 'Let Property' et quatre méthodes publiques. Elle possède aussi une méthode privée.

Détaillons un peu.

La propriété EstConnecté permet de vérifier si le composant est connecté au serveur.

La méthode Deconnexion déconnecte le composant du serveur

La méthode ChaineProc renvoie le texte de la procédure stockée dont le nom est passé en paramètre.

La méthode ListeProc renvoie un variant contenant un tableau de chaînes représentant la liste des procédures stockées d'une base

La méthode Connexion attend les paramètres nécessaires à l'établissement de celle-ci.

Enfin il existe une procédure privée Terminate qui libère les objets SQLDMO utilisés.

Je compile ma DLL et j'obtiens donc un fichier StoreProc.dll.

## *Enregistrer le composant*

Tout d'abord j'enregistre mon composant. Ce ne sont pas les moyens qui manquent, dans cet exemple j'exécute la ligne de commande "regsvr32 C:\JMARC\Sources\stproc\storeproc.dll".

Maintenant je vais voir ce qu'il s'est passé dans la base de registre.

Dans la catégorie ClasseRoot\CLSID je trouve :

```
[HKEY_CLASSES_ROOT\CLSID\{8F7F2A40-5D6F-4C5E-8359-D9B965297EAB}]
@="StoreProc.clsStoreProc"
"AppID"="{8F7F2A40-5D6F-4C5E-8359-D9B965297EAB}"

[HKEY_CLASSES_ROOT\CLSID\{8F7F2A40-5D6F-4C5E-8359-
D9B965297EAB}\Implemented Categories\{40FC6ED5-2438-11CF-A3DB-080036F12502}]

[HKEY_CLASSES_ROOT\CLSID\{8F7F2A40-5D6F-4C5E-8359-
D9B965297EAB}\InprocServer32]
@="C:\\JMARC\\Sources\\stproc\\storeproc.dll"
"ThreadingModel"="Apartment"
```

```
[HKEY_CLASSES_ROOT\CLSID\{8F7F2A40-5D6F-4C5E-8359-D9B965297EAB}\ProgID]
@="StoreProc.clsStoreProc"

[HKEY_CLASSES_ROOT\CLSID\{8F7F2A40-5D6F-4C5E-8359-
D9B965297EAB}\Programmable]

[HKEY_CLASSES_ROOT\CLSID\{8F7F2A40-5D6F-4C5E-8359-D9B965297EAB}\TypeLib]
@="{94B877DA-7F33-405F-B14A-190112750406}"

[HKEY_CLASSES_ROOT\CLSID\{8F7F2A40-5D6F-4C5E-8359-
D9B965297EAB}\VERSION]
@="5.0"
```

Ces suites de caractères sont des identificateurs uniques (GUID).

Vous voyez qu'on y trouve le chemin de la DLL, l'identificateur unique de la classe (CLSID) et l'identificateur de la bibliothèque de type.

Si je poursuis ma recherche je retrouve dans la catégorie Interface

```
[HKEY_CLASSES_ROOT\Interface\{2A08C76E-4C84-42F0-9E16-98F38423596C}]
@="clsStoreProc"

[HKEY_CLASSES_ROOT\Interface\{2A08C76E-4C84-42F0-9E16-
98F38423596C}\ProxyStubClsid]
@="{00020424-0000-0000-C000-000000000046}"

[HKEY_CLASSES_ROOT\Interface\{2A08C76E-4C84-42F0-9E16-
98F38423596C}\ProxyStubClsid32]
@="{00020424-0000-0000-C000-000000000046}"

[HKEY_CLASSES_ROOT\Interface\{2A08C76E-4C84-42F0-9E16-98F38423596C}\TypeLib]
@="{94B877DA-7F33-405F-B14A-190112750406}"
"Version"="5.0"
```

Là je trouve les identificateurs nécessaires pour gérer le Marshalling DCOM (Proxy et Stub)

Notez que mon composant ne gère qu'une interface. Cela est logique puisque je n'ai pas décrit d'interface particulière, VB6 a généré une interface par défaut. Cet identificateur d'interface (IID) est celui qui va être appelé par la méthode QueryInterface de IUnknown.

Toujours dans HKEY_CLASSES_ROOT il existe une entrée StoreProc.clsStoreProc qui pointe vers le CLSID que nous avons vu au-dessus.

Il existe encore une entrée pour la bibliothèque de type

```
[HKEY_CLASSES_ROOT\TypeLib\{94B877DA-7F33-405F-B14A-190112750406}\5.0]
@="StoreProc"
[HKEY_CLASSES_ROOT\TypeLib\{94B877DA-7F33-405F-B14A-190112750406}\5.0\0]

[HKEY_CLASSES_ROOT\TypeLib\{94B877DA-7F33-405F-B14A-
190112750406}\5.0\0\win32]
@="C:\\JMARC\\Sources\\stproc\\storeproc.dll"

[HKEY_CLASSES_ROOT\TypeLib\{94B877DA-7F33-405F-B14A-
190112750406}\5.0\FLAGS]
@="0"

[HKEY_CLASSES_ROOT\TypeLib\{94B877DA-7F33-405F-B14A-
190112750406}\5.0\HELPDIR]
@="C:\\JMARC\\Sources\\stproc"
```

Ca en fait des entrées pour un composant aussi basique. Vous voyez là un des problèmes du modèle COM qui nécessite des enregistrements complexes pour fonctionner.

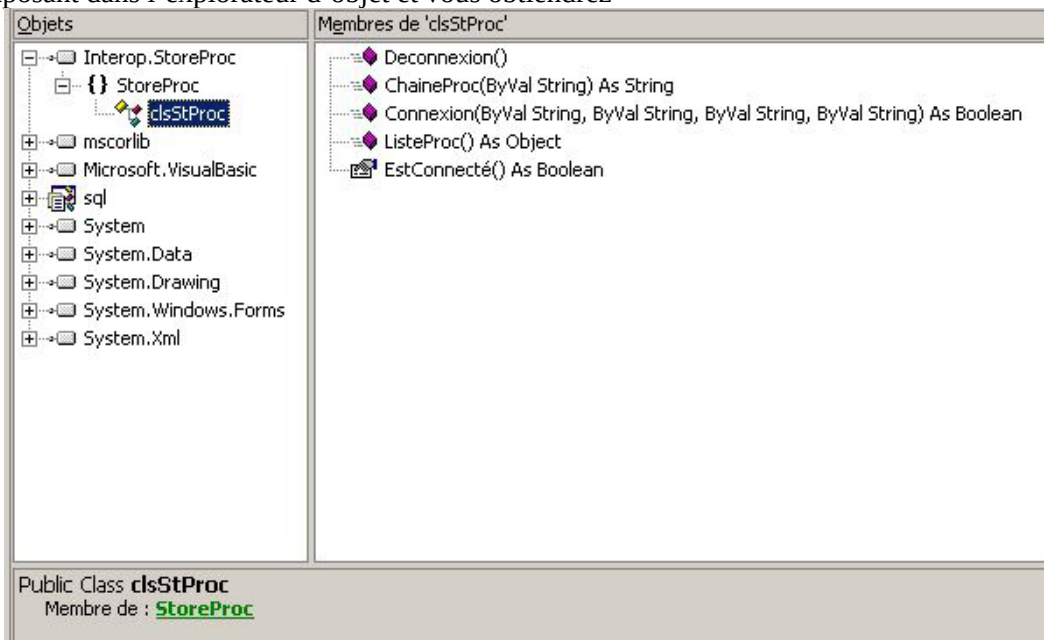
## *Générer le projet managé*

Je vais donc maintenant écrire mon programme VB.NET qui va utiliser mon composant.

Je crée donc un nouveau projet Windows.Form. Sur la Feuille j'ajoute une zone de liste et un textBox. Je définis sa propriété Multiline à vrai et je lui définis une ScrollBar verticale. J'insère aussi une connexion SQL à partir de mon explorateur de serveur.

Dans le menu projet, j'ajoute une référence à ma DLL. (Ajouter une référence – Onglet COM – Sélectionner StoreProc).

Dans l'explorateur de solution, dans les références je vois apparaître une référence StoreProc (de son vrai nom interop.storeproc.dll). Si vous êtes d'un naturel curieux vous pouvez aller voir notre composant dans l'explorateur d'objet et vous obtiendrez




Je retrouve bien mes quatre procédures et ma propriété. Notons que ce que nous voyons là est le PIA et que comme je vous l'ai dit sur le marshalling interop, la méthode ListeProc renvoie maintenant 'Object' au lieu de Variant.

J'écris maintenant mon code de manipulation.

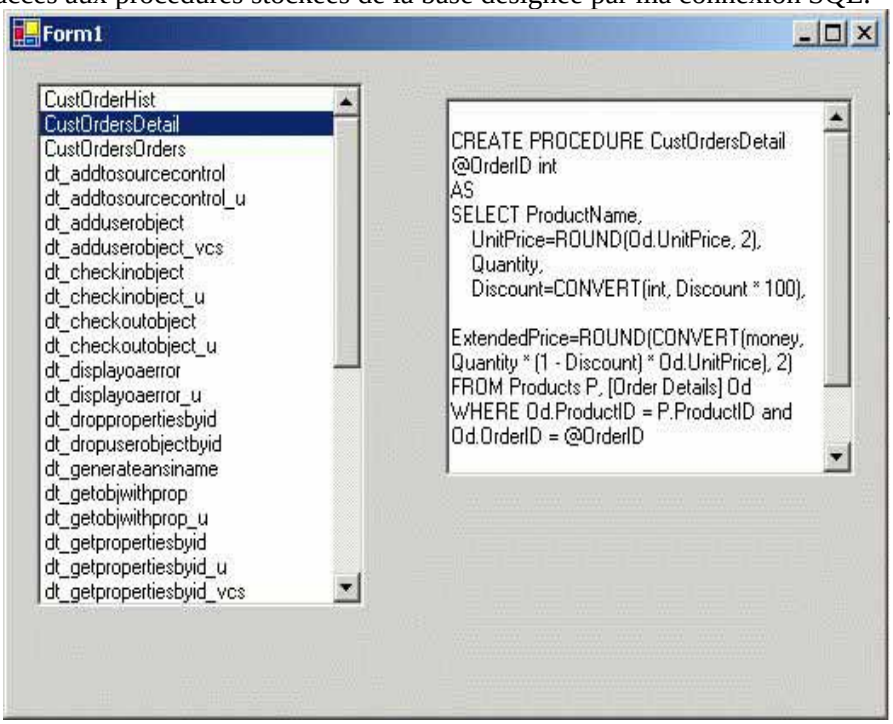
```
Private MaProc As New StoreProc.clsStProc
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

        Dim Liste As Object
        Dim Recup As Boolean =
MaProc.Connexion(Me.SqlConnection1.DataSource,
Me.SqlConnection1.Database, "sa", "monpasse")
        Liste = MaProc.ListeProc
        If IsArray(Liste) Then
            Dim cmpt As Long
            For cmpt = LBound(Liste) To UBound(Liste)
                Me.ListBox1.Items.Add(Liste(cmpt))
            Next
        End If

    End Sub

    Private Sub ListBox1_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles ListBox1.Click
        Me.TextBox1.Text =
MaProc.ChaineProc(Me.ListBox1.SelectedItem)
    End Sub
```

Voilà, j'ai accès aux procédures stockées de la base désignée par ma connexion SQL.




Ce n'est pas bien compliqué, mais poursuivons un peu.

# *Discussions complémentaires*

## Un composant mal conçu

Vous n'avez peut être pas fais attention au code, mais le composant que j'ai conçu est un pur chef d'œuvre de composant inutilisable. Si j'étais professeur, je pourrais faire un exercice avec « trouver les erreurs de conception du composant StoreProc » que nous allons voir maintenant.

Son défaut général est déjà évident, il n'est pas correctement documenté. Il est pratiquement inutilisable pour qui n'a pas lu le code source. En fait, tout commence mal dès l'utilisation de la méthode Connexion.

```
Public Function Connexion(ByVal Serveur As String, ByVal Catalogue
As String, ByVal User As String, ByVal Password As String) As
Boolean

    On Error GoTo Echec_Connect
    Set MonServeur = New SQLDMO.SQLServer
    MonServeur.LoginSecure = True
    MonServeur.Connect Serveur, User, Password
    Set MaBase = New SQLDMO.Database
    Set MaBase = MonServeur.Databases(Catalogue)
    mvarEstConnecté = True
    Connexion = True
    Exit Function
Echec_Connect:
    Err.Clear
    Connexion = False
    On Error GoTo 0

End Function
```

Dans cette fonction, il y a un problème de conception. Si les noms des paramètres sont clairs, il n'en est pas de même de la valeur retournée. En effet il n'est pas évident de savoir que si la méthode renvoie faux, c'est que la connexion a échoué. Il y a surtout une grave faute de logique.

La connexion peut échouer pour de nombreuses raisons, toutes n'ayant pas les mêmes implications. Si l'échec est dû à un TimeOut on ne peut pas y faire grand-chose, par contre s'il s'agit d'un mauvais paramètre, le code client devrait pouvoir remédier à cela. Telle que la fonction est conçue, tous les échecs se résument à une réponse Oui/Non.

Nous sommes là dans un cas où l'erreur doit être propagée vers le code client. La gestion de l'erreur à ce niveau est une erreur. On peut d'ailleurs étendre ce raisonnement à tous les composants ayant un point d'entrée privilégié attendant des paramètres.

Dans un autre ordre d'idée, il vaut mieux éviter de transmettre des booléens. En effet l'interprétation de la valeur Vrai (<>0, 1, -1 ) peut poser des problèmes.

Ma nouvelle fonction sera donc une procédure.

Ma deuxième erreur est malheureusement assez fréquente. De manière générale on évite toujours de renvoyer des variants ou des types génériques. En effet la récupération pose souvent des problèmes et demande une manipulation assez lourde pour identifier le sous-type du variant. De plus, le nom de la méthode est ambigu. Ce qui fait qu'on ne peut pas savoir à priori que le variant renvoyé doit être un tableau. Pour éviter ce problème, il faut changer le nom de la procédure, et renvoyer un tableau.

Enfin le troisième défaut porte sur l'instanciation. S'il peut paraître logique d'appeler la méthode connexion avant de travailler avec le composant, rien ne l'impose. Or c'est dans cette méthode que l'on instancie l'objet SQLServer. Tant qu'on n'est pas entré dans la méthode connexion, toutes les autres méthodes vont déclencher une erreur du fait de la non existence des objets. Pour pallier à cela, je devrais écrire plutôt mon composant ainsi.

```
'variables locales de stockage des valeurs de propriétés
Private mvarEstConnecté As Boolean 'copie locale
Private MonServeur As SQLDMO.SQLServer
Private MaBase As SQLDMO.Database

Public Property Get EstConnecté() As Boolean
    If MonServeur Is Nothing Then
        mvarEstConnecté = False
    Else
        mvarEstConnecté                                                    =
MonServeur.VerifyConnection(SQLDMOConn_CurrentState)
    End If
    EstConnecté = mvarEstConnecté
End Property

Public Sub Deconnexion()

    If Me.EstConnecté Then MonServeur.DisConnect

End Sub

Public Function ChaineProc(ByVal NomProc As String) As String

    If Me.EstConnecté Then
        On Error Resume Next
        ChaineProc = MaBase.StoredProcedures(NomProc).Text
        If Err.Number > 0 Then
            MsgBox "la procédure " & NomProc & " n'existe pas",
            vbCritical + vbOKOnly, "ERREUR"
        End If
        On Error GoTo 0
    Else
        PasConnecté
    End If
End Function
```

```
Public Function TableauNomProc() As String()
    Dim Procedure As SQLDMO.StoredProcedure, NbProcedure As Integer,
cmpt As Long, TabChaine() As String

    If Me.EstConnecté Then
        NbProcedure = MaBase.StoredProcedures.Count
        If NbProcedure > 0 Then
            ReDim TabChaine(0 To NbProcedure - 1)
            For cmpt = 0 To NbProcedure - 1
                TabChaine(cmpt) = MaBase.StoredProcedures(cmpt +
1).Name
            Next cmpt
            TableauNomProc = TabChaine
        Else
            MsgBox "Il n'y a pas de procédures stockées dans le
catalogue" & MaBase.Name, vbInformation + vbOKOnly, "ERREUR"
        End If
    Else
        PasConnecté
    End If

End Function

Public Sub Connexion(ByVal Serveur As String, ByVal Catalogue As
String, ByVal User As String, ByVal Password As String)

    Set MonServeur = New SQLDMO.SQLServer
    MonServeur.LoginTimeout = 10
    MonServeur.LoginSecure = True
    MonServeur.Connect Serveur, User, Password
    Set MaBase = New SQLDMO.Database
    Set MaBase = MonServeur.Databases(Catalogue)
    mvarEstConnecté = True

End Sub

Private Sub PasConnecté()
    MsgBox "Le composant n'est pas connecté à une base SQL-Server",
vbCritical + vbOKOnly, "ERREUR"
End Sub

Private Sub Class_Terminate()

    If Me.EstConnecté Then MonServeur.DisConnect
    Set MaBase = Nothing
    Set MonServeur = Nothing

End Sub
```

Voilà. Si vous êtes vigilants, vous verrez qu'il reste une petite faute.

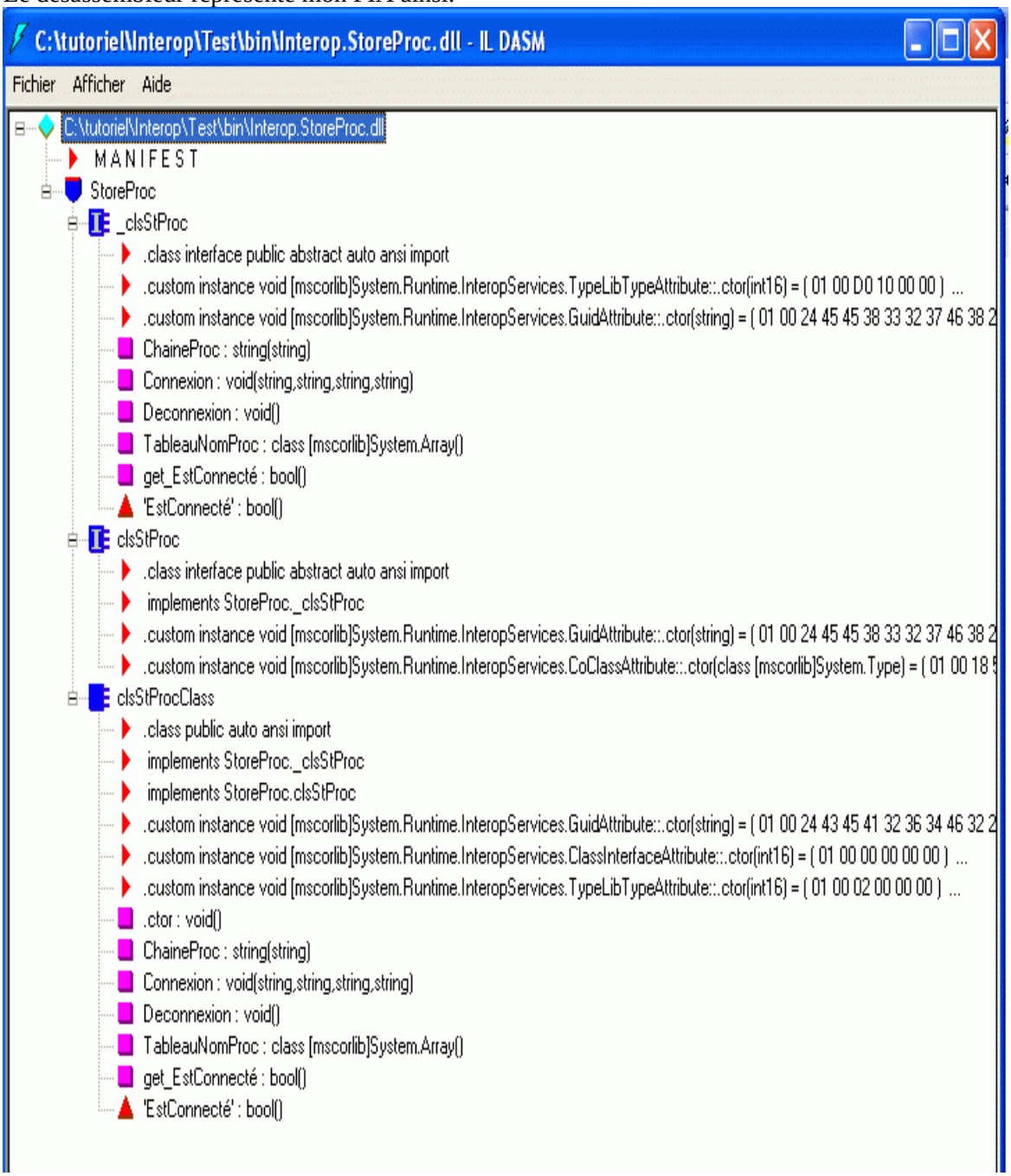
Il est évident que le code du client doit changer puisque j'ai changé mon composant. Mais nous allons en profiter pour manipuler un peu le Wrapper afin de voir comment il fonctionne.

## Voir le PIA.

Visual studio permet d'utiliser un utilitaire intéressant, le MSIL Disassembler. On le trouve dans le sous répertoire Bin du SDK (ildasm.exe).

Allons voir à quoi ressemble notre nouvel assembly interop.storeproc.dll.

Le désassembleur représente mon PIA ainsi.




Vous avez là d'un coup d'œil la réalité du Wrapper.

Le PIA contient l'ensemble des GUID nécessaires à la manipulation, et l'ensemble des types utilisable, vu du coté framework.

Je ne vais pas détailler ici ce que nous voyons puisque si vous avez à peu près suivi mes explications vous devez le comprendre.

Notez juste qu'il y a conversion de type pour TableauNomProc puisqu'un tableau de chaîne n'est pas blittable.

## Générer le PIA par le code

Comme je vous l'ai dit, il est possible d'utiliser la classe TypeLibConverter pour générer le PIA par le code.

```
Imports System.Runtime.InteropServices
Imports System.Reflection
Imports System.Reflection.Emit

Public Class CreatePIA
    Private Enum RegKind
        RegKind_Default = 0
        RegKind_Register = 1
        RegKind_None = 2
    End Enum 'RegKind

    <DllImport("oleaut32.dll",             CharSet:=CharSet.Unicode,
PreserveSig:=False)> _
    Private  Shared  Sub  LoadTypeLibEx(ByVal  strTypeLibName  As
[String],         ByVal         regKind         As         RegKind,
<MarshalAs(UnmanagedType.Interface)> ByRef typeLib As [Object])
    End Sub

    Shared Sub Main()

        Dim typeLib As [Object]
        LoadTypeLibEx("d:\user\jmarc\sources\storeproc.dll",
RegKind.RegKind_None, typeLib)

        If typeLib Is Nothing Then
            Console.WriteLine("LoadTypeLibEx failed.")
            Return
        End If

        Dim converter As New TypeLibConverter
        Dim eventHandler As New ConversionEventHandler
        Dim   asm   As   System.Reflection.Emit.AssemblyBuilder   =
converter.ConvertTypeLibToAssembly(typeLib,   "StoreProc.dll",   0,
eventHandler, Nothing, Nothing, Nothing, Nothing)
        asm.Save("StoreProc.dll")

    End Sub
End Class

Public Class ConversionEventHandler
    Implements
System.Runtime.InteropServices.ITypeLibImporterNotifySink

    Public  Sub  ReportEvent(ByVal  eventKind  As  ImporterEventKind,
ByVal  eventCode  As  Integer,  ByVal  eventMsg  As  String)  Implements
ITypeLibImporterNotifySink.ReportEvent
        ' Evenement pour les avertissement
    End Sub

    Public  Function  ResolveRef(ByVal  typeLib  As  Object)  As
[Assembly] Implements ITypeLibImporterNotifySink.ResolveRef
        ' renvoie l'assembly correspondant à la bibliothèques de
type passée en paramètre
        Return Nothing
    End Function
End Class
```

## Manipulez correctement le composant

Le code de manipulation doit donc gérer les erreurs qui peuvent se propager à partir du composant.

```
Private VisuProc As StoreProc.clsStProc
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        VisuProc = New StoreProc.clsStProc
        Try
            VisuProc.Connexion("NOM-SZ71L7KTH17\TUTO", "northwind",
"sa", "monpasse")
            Dim TabProcedure As Array = VisuProc.TableauNomProc
            If TabProcedure.GetLength(0) > 0 Then
                Dim cmpt As Integer
                For cmpt = TabProcedure.GetLowerBound(0) To
TabProcedure.GetUpperBound(0)

Me.ListBox1.Items.Add(TabProcedure.GetValue(cmpt))
                Next
            End If
        Catch ex As System.Runtime.InteropServices.COMException
            Select Case ex.ErrorCode.ToString
                Case -2147199728
                    MessageBox.Show("Mauvais nom de catalogue")

                Case -2147221504
                    MessageBox.Show("Mauvais nom de serveur")

                Case Else
                    MessageBox.Show("Utilisateur ou mot de passe
invalide")
            End Select
        Catch ex As Exception
            MessageBox.Show("La connexion a échouée")
        End Try
    End Sub

    Private Sub ListBox1_SelectedIndexChanged(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
ListBox1.SelectedIndexChanged
        Me.TextBox1.Text =
VisuProc.ChaineProc(Me.ListBox1.SelectedItem)
    End Sub

    Private Sub Form1_Closing(ByVal sender As Object, ByVal e As
System.ComponentModel.CancelEventArgs) Handles MyBase.Closing
        VisuProc.Deconnexion()
    End Sub
```

Notez que le code est plus aisément compréhensible du fait du passage d'un tableau.

# Conclusion

Nous n'avons fait là qu'une visite superficielle de l'utilisation des composants COM avec du code managé. Je n'ai pas détaillé le cas inverse, c'est à dire l'utilisation d'un composant managé dans un environnement non managé, car je n'ai encore jamais rencontré le cas.

Bonne programmation