

# *Introduction à COM*

*(White paper)*

→ Sujet :

*Ce document présente les fondements de COM. Il n'a pas pour prétention de se substituer à un cours complet , mais plutôt d'introduire les notions principale de l'architecture phare de Microsoft.*

→ A Qui s'adresse ce document :

*Ce document s'adresse aux développeurs Windows qui désirent s'initier à COM. J'entends par développeurs Windows des personnes qui programment avec Visual C++, ou en C avec le SDK Win32. Mis à part pour des raisons de culture générale, des développeurs VB n'y trouveraient pas leur compte ... en effet, cet outillage masque complètement la réalité de COM !*

→ Quel niveaux ?

*Afin de bien appréhender ce document il est indispensable d'avoir de bonnes notions du C++.*



## *Table des matières*

1	Les composants	3
1.1	Un peu d'histoire ...	3
1.2	Introduction à COM	3
1.3	Création de composant	4
1.3.1	Introduction	4
1.3.2	Gestion d'une seconde interface	6
2	Les fondements de COM	7
2.1	Terminologie & Représentation	7
2.2	Le rôle de la Registry et les CLSID	8
2.3	Utiliser un Objet COM	8
3	Les Interfaces COM	10
3.1	L'interface IUnknown	10
3.1.1	Le comptage de référence	10
3.1.2	Le changement d'interface	11
3.2	L'interface IClassFactory	12
3.2.1	Description des méthodes	12
3.2.2	Synoptique de fonctionnement	12
4	Automation	14
4.1	Introduction	14
4.2	IDispatch	14
4.3	Le type Variant	16
5	Les types d'interfaces COM	18
5.1	Le marshalling	18
5.2	displInterface	18
5.3	Custom Interface	18
5.4	Interfaces duales	19

# 1 Les composants

## 1.1 Un peu d'histoire ...

Inutile de préciser que **COM/DCOM** est une technologie (ou plutôt dirais-je une méthode de création de composant) conçue par Microsoft à la fin des années 1980. Mais si l'architecture est maintenant robuste est entièrement spécifiée, il n'en a pas toujours été ainsi car COM comme toutes les technologies pérennes est passé par plusieurs stades successifs avant d'arriver à une certaine maturité (technique comme marketing !).

Tout a commencé par l'équipe de développeurs Excel, en 1989 (si je ne m'abuse !) qui spécifie et implémente **DDE** (protocole basé sur des messages Windows) afin de permettre à d'autres applications de communiquer avec Excel.

Là dessus l'équipe de développeurs Word, en 1992, se disent qu'ils peuvent faire mieux et intégrer des applications dans Word (comme des images par exemples !) et ils implémentent **OLE** (protocole toujours basé sur des messages Windows).

Mais comme, on n'a pas encore fait le tour de la famille Office, c'est l'équipe PowerPoint qui, en 1993, améliore les spécifications de OLE en créant **OLE2**. Ce dernier protocole n'étant plus fondé sur des messages Windows mais sur des pointeurs de fonctions ... COM est né !

Pour la petite histoire marketing, OLE2 est rebaptisé OLE parce que Microsoft a décidé que « marketingement » parlant il n'y aura pas de OLE3 !

Surviennent successivement : **DCOM** (Distributed COM) en 1995 puis en 1996 naît la terminologie **ActiveX** qui englobe (enfin là dessus rien n'est sûr car même chez Microsoft tout le monde n'est pas d'accord ... mais bon ce n'est qu'une couche marketing qui est fondée sur la même technologie : COM) COM et Internet.

Ensuite c'est la librairie **ATL** qui est développée afin de permettre la création de composants petits par la taille mais grand par l'efficacité, enfin en 1998 **MTS** introduit la notion de serveur transactionnel.

Pour finir (pour l'instant) 2000 est l'arrivée de **COM+** !

Objectifs de COM

Voici les principaux objectifs de COM (Cf. Spécifications de COM) :

- Définir un protocole logiciel de création de composant.
- L'accès aux composants ainsi que leur développement (pourvu que le langage ait accès aux pointeurs de fonctions) est indépendant du langage de programmation. Il est possible d'utiliser par exemple : C, C++, Java, Visual Basic Etc.
- La localisation du composant doit être transparente (un client doit pouvoir faire appel à un composant sans savoir où il se trouve : répertoire, autre machine, Etc.)
- Réduire le problème de versionning. En empêchant les petits malins d'accéder grâce aux pointeurs aux membres privés d'une classe. En fournissant des interfaces immuables !

## 1.2 Introduction à COM

Bon, pour commencer par le début ... Qu'est-ce que c'est donc qu'un composant ?

Et bien on peut dire qu'un composant est en fait << un fournisseur de service logiciel qui permet à une application de s'affranchir de certaines opérations déjà modélisées et réalisées >>. Voilà pour une



définition au sens large ... pour être plus précis un composant est en fait un *fournisseur d'objet (comme une classe) compilé* dans une DLL ou un Exécutable.

En COM/DCOM on distingue trois type de composants :

- Les **DLL** (*in-process*) : qui sont des composant métiers la plupart du temps
- Les **EXE** (*out-of-process*)
- Les **OCX** (*in-process*) : qui sont en réalité des DLL (renommées en OCX) intégrant la notion d'IHM. Ces composant permettent la création d'objets graphiques.

Les composants COM reposent:

- Sur la notion d'interface<sup>1</sup> par laquelle les clients peuvent accéder. Ces interfaces sont immuables et offrent un contrat de service entre le client et le fournisseur.
- Sur des numéros uniques appelés **CLSID**, **UUID** et **GUID**(uniques), qui permettent de localiser un composant grâce à la **Registry**. Ainsi chaque composant possède son CLSID unique, chaque interface possède son GUID unique Etc.

## 1.3 Création de composant

### 1.3.1 Introduction

Pour comprendre l'esprit de COM, on ne va pas se jeter tête baissée dans la programmation COM avec les moyens (bibliothèques fournies par Microsoft) ... quelque part ça serait trop facile, et puis ça ne permettrait pas de bien comprendre comment et pourquoi COM a été conçu ! non, on va créer un composant (pas COM mais suivant son principe !) de A à Z en C++ pur. On verra, plus tard, comment utiliser ces bibliothèques (Cf. les chapitres suivants !).

Pour expliquer comment est venue la naissance de composant et plus particulièrement de COM, nous allons partir d'une classe que nous allons progressivement transformer en composant :

```
class CPhone
{
    private:
        long m_lngNumber;
    public:
        bool Call();
        void Ring(int _iDelay);
};
```

Intégrons la notion d'interface. Cette manipulation a pour but de fournir à un utilisateur de notre classe l'interface d'un objet et non l'objet directement. Ainsi on crée une indirection avec des pointeurs de fonction (puisque qu'une interface n'est en fait qu'une **VTABLE** = liste de pointeurs de fonction). Nous avons maintenant deux classes :

```
class IPhone
{ // Classe Virtuelle pure
    public:
        virtual bool Call() = 0;
        virtual void Ring(int _iDelay) = 0;
};
```

---

<sup>1</sup> Note (pour les développeurs Visual Basic) : Attention VB confond Objet COM et Interface COM !



```
class CPhone : IPhone
{
    private:
        long m_IngNumber;
    public:
        bool Call() { /* Surcharge */ };
        void Ring(int _iDelay) { /* Surcharge */ };
};
```

Afin de fournir à notre client potentiel une interface et non l'objet nous devons créer une fonction quiinstanciera l'objet et ne rendra que l'interface IPhone (cette action aura aussi pour avantage de nous affranchir de l'opérateur new propre au langage C++).

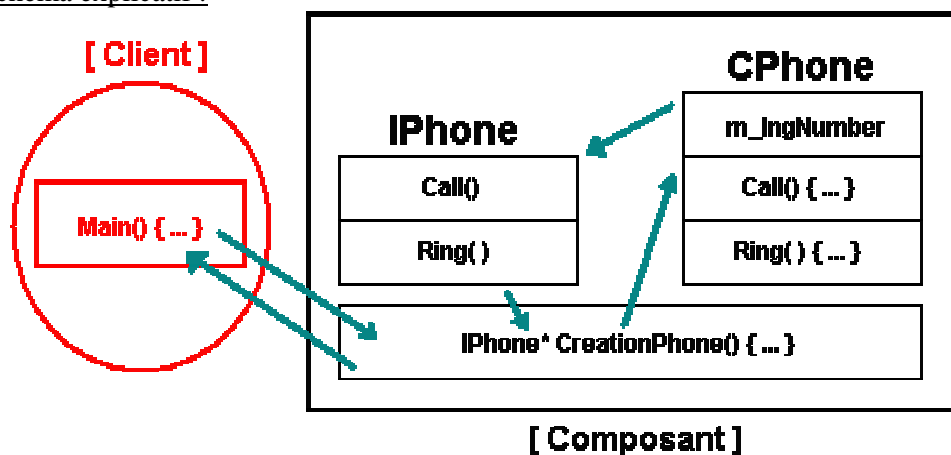
```
IPhone* CreatePhone()
{
    CPhone *phone = new CPhone();
    return (IPhone*)phone;
}
```

Un client pourra demander la création d'un objet de cette façon :

```
void main()
{
    IPhone* obj = CreatePhone();
    obj->Call();
}
```

C'est cette fonction, et seulement cette fonction (enfin avec l'interface IPhone) qui sera accessible par un client ! Voilà pour le principe global (et grandement simplifié) de COM. Pour résumer le composant (prenons une DLL pour simplifier) sera compilé avec les deux classes IPhone et CPhone ainsi qu'avec la fonction exportée CreationPhone().

Voilà un schéma explicatif :



Pour y accéder le fournisseur devra fournir :

- La DLL compilée.
- Un fichier header (\*.h) contenant la description de la classe virtuelle pure IPhone.



### 1.3.2 Gestion d'une seconde interface

Pour corser le tout, nous allons maintenant apporter la notion d'interface à notre mini-composant. Cette opération est très utile et permettra à un objet d'être vu de deux manière différentes. Pour cela nous allons créer deux classes virtuelles pure et une classe (l'objet réel) qui dérivera de ces deux interfaces. Nous utiliserons l'héritage multiple mais on aurait pu aussi utiliser des sous-classes amis encapsulées.

```
class iPhone1
{ // Classe Virtuelle pure
public:
    virtual bool Call() = 0;
};

class iPhone2
{ // Classe Virtuelle pure
public:
    virtual void Ring(int _iDelay) = 0;
};

class CPhone : iPhone1, iPhone2
{
private:
    long m_lngNumber;
public:
    bool Call() { /* Surcharge */ };
    void Ring(int _iDelay) { /* Surcharge */ };
};
```

La fonction d'instantiation ainsi que le programme doivent changer quelque peu pour prendre en compte les deux interfaces :

```
void* CreatePhone(int numInterface)
{
    if (numInterface == 1)
        return (void*)(iPhone1 *)new CPhone();
    else if (numInterface == 2)
        return (void*)(iPhone2 *)new CPhone();
    else
        return NULL;
}

void main()
{
    iPhone2* obj2 = (iPhone2*)CreatePhone(2);
    iPhone1* obj1 = (iPhone1*)CreatePhone(1);
    obj2->Ring(1);
    obj1->Call();
}
```

Dorénavant obj1 ne voit que la méthode Call() et obj2 ne voit que la méthode Ring() et pourtant ces deux objets proviennent de la même classe de base CPhone.

Cette gestion d'interface permet de gérer plus facilement l'immuabilité des interface COM. En effet dès qu'une modification doit être effectuée sur un composant, au lieu de modifier le code (ce qui est interdit ... vu que les interface - VTABLE- sont immuables) on crée une autre interface !).

## 2 Les fondements de COM

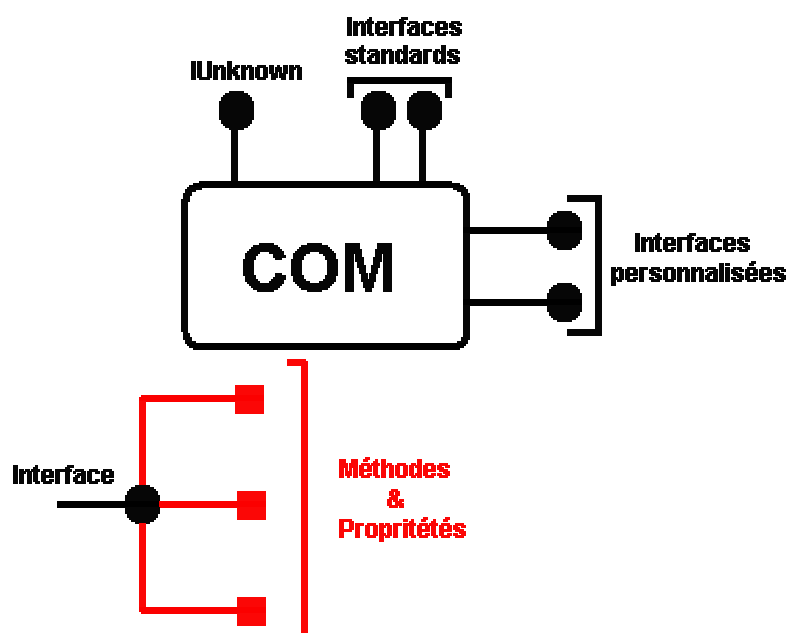
Voilà nous venons de créer un mini composant, qui introduit l'architecture (ou plutôt le protocole logiciel) COM spécifié par Microsoft, maintenant nous allons voir comment COM est réellement conçu ... Alors accrochez-vous à vos baskets car on va mettre les mains dans le cambouis !

### 2.1 Terminologie & Représentation

Tout d'abord on va passer en revue les termes qui seront abordés tout au long de ce chapitre :

<i>Terminologie</i>	<i>Explication</i>
<b>COM</b>	Component Object Model
<b>DCOM</b>	Distributed COM
<b>OLE</b>	Object Linking And Embedding
<b>Interface</b>	Une interface est en faite une classe virtuelle pure. Cette dernière permet la séparation de l'interface (méthode et propriétés de la classe) de son implémentation (qui a lieu dans une autre classe qui en dérive).
<b>CLSID</b>	Nombre sur 128 bits qui référence de manière unique un objet COM
<b>ProgID</b>	Nom commun (pas forcément unique !) sous forme de chaîne de caractère caractérisant un objet COM.
<b>UNICODE</b>	Mode de codage de caractères sur 2 Octets ( contrairement aux codes ASCII sur 1 Octet)
<b>TypeLibrary</b>	Le rôle de la typeLibrary est similaire à un fichier header (*.h) en C++, il contient la description des méthodes et propriétés du composant sous forme binaire et peut être contenu dans le composant même ou séparément (*.tlb).

D'autre part on représente un composant COM de cette manière :





## 2.2 Le rôle de la Registry et les CLSID

Afin de bien comprendre comment COM s'affranchit de la localisation des composant, il faut se pencher de plus prêt vers la Registry Windows. Or, comme nous l'avons vu, les CLSID représentant de manière unique (l'algorithme dépend du numéro MAC de la carte réseau, de l'heure Etc.) un composant COM ... Et bien la base de registre contient en fait une liste de tous les composants enregistrés sur la machine (enfin de leur CLSID<sup>2</sup> et puis d'autres informations).

Tout cela nous amène à un nouveau point essentiel ... il faut dire explicitement à la machine que tel composant de tel type est présent à tel endroit ! c'est ce que l'on appelle **l'Enregistrement/Désenregistrement du composant** et d'après les spécifications de COM c'est au composant d'effectuer le travail d'écriture dans la registry (Enregistrement / Désenregistrement).

Dans le cas d'une DLL il faudra donc prévoir deux fonction exportées réalisant ces opération : **DllRegisterServer** et **DllUnregisterServer**. Afin d'enregistrer le composant il ne restera plus qu'à appeler ces deux fonctions à l'aide de l'utilitaire REGSVR32.EXE fournit par Microsoft. Pour un EXE il faudra prévoir la gestion des paramètres de commande (**/REGSERVER** & **/UNREGSERVER**).

Il existe un autre moyen de retrouver un composant ... grâce à son ProgID, qui est en fait une chaîne de caractère plus parlante ... l'inconvénient c'est qu'elle n'est pas unique ! il faut donc une table de correspondance entre ce ProgID et le CLSID. C'est encore une fois de plus le rôle de la registry qui stocke directement sous **HKEY\_CLASSES\_ROOT** ces ProgIDs sous forme de clefs, une sous-clef CLSID<sup>3</sup> pointe sur le CLSID correspondant (la fonction **CLSIDFromProgID(...)** retourne le CLSID correspondant à un ProgID).

Le rôle de la registry ne s'arrête pas là, car elle doit aussi indiquer de quelle sorte de composant il s'agit. Pour cela le CLSID peut contenir d'autres sous clef comme :

- **InprocServer32** : Spécifie que le composant est une DLL.
- **LocalServer32** : Spécifie que le composant est un EXE.
- **RemoteServer32** : Spécifie que le composant se trouve sur un ordinateur distant.

Voilà pour les principales sous-clef des CLSID, mais il peut en contenir d'autre précisant par exemple le mode de threading utilisé, le CLSID de la TypeLibrary Etc.

## 2.3 Utiliser un Objet COM

Dans cette partie nous allons créer un Client COM qui instancie un composant COM existant. Avant toute chose il est à préciser que par convention **TOUTES les methodes et propriétés COM** (une propriété COM est en fait composé de deux methodes en lecture et ecriture) retournent une variable de type **HRESULT**<sup>4</sup> (qui est en fait un entier long).

D'autre part avant d'instancier un objet COM il faut initialiser les librairies COM ... c'est le rôle de **CoInitialize()** et **CoInitializeEx()** puis une fois que l'on a utilisé ses objets COM il faut cloturer cette initialisation avec la fonction<sup>5</sup> **CoUninitialize()**.

<sup>2</sup> Il existe un utilitaire nommé **GUIDGEN.EXE** qui génère ces CLSID

<sup>3</sup> Les CLSID sont stockés dans la registry sous la clef **HKEY\_CLASSES\_ROOT\CLSID**

<sup>4</sup> Cette variable selon sa valeur indique que la méthode a été effectuée avec succès ou non et précise le type d'erreur apparue.

<sup>5</sup> Quasiment toutes les fonctions COM commencent par **Co**.





Voici un exemple d'utilisation :

```
#include <windows.h>
#include <comdef.h>
const IID IID_IRadio =
{0xAA74361B,0x4C11,0x11D4,{0xAA,0x54,0x00,0x50,0xDA,0x68,0xAE,0x82}};
const CLSID CLSID_Radio =
{0xAA74361C,0x4C11,0x11D4,{0xAA,0x54,0x00,0x50,0xDA,0x68,0xAE,0x82}};

void main()
{
    CoInitialize(NULL);

    HRESULT hr;
    IRadio* pRadio;
    _bstr_t strStation("TEST");

    hr = CoCreateInstance(CLSID_Radio,      // CLSID du composant
                        NULL,              // --> Sert pour l'aggrégation !
                        CLSCTX_ALL,        // indique quoi rechercher (dll, exe, distant)
                        IID_IRadio,        // interface
                        (void**)&pRadio);  // objet retourné

    if (FAILED(hr)) return;

    // Utilisation du composant ...

    pRadio->Release();
    CoUninitialize();
}
```

Dans l'exemple ci-dessus on utilise la fonction **CoCreateInstance(...)** qui permet l'instanciation d'objets COM (c'est un peu l'équivalent de notre **CreatePhone()** de la page 1).

## 3 Les Interfaces COM

### 3.1 L'interface IUnknown

Nous avons vu que COM reposait sur la notion d'interface (qui est en fait une table de pointeurs de fonctions) ... Et bien en COM il y en a une qui est absolument obligatoire (c'est en quelque sorte ce qui fait l'identité d'un objet COM) : **IUnknown** : tous les objets COM doivent en dériver !

Voici son prototype :

```
struct IUnknown {  
public:  
    virtual HRESULT _stdcall QueryInterface(REFIID riid, void **ppv) = 0;  
    virtual ULONG _stdcall AddRef() = 0;  
    virtual ULONG _stdcall Release() = 0;  
};
```

→ Cette classe <sup>6</sup> à deux objectifs : Effectuer le comptage de référence et permettre de changer d'interface.

#### 3.1.1 Le comptage de référence

Pourquoi effectuer un comptage de référence ? et puis d'abord qu'est-ce que c'est qu'un comptage de référence ? voilà deux questions auxquelles nous allons répondre. Tout d'abord, qu'elle est la problématique ? et bien, nos composants COM ont un petit soucis (coté serveur) ... on ne sait pas quand les supprimer de la mémoire (et oui ce n'est pas le travail de COM).

En effet, il est tout à fait possible que plusieurs clients fassent appel à nos composant ! notre composant sera donc chargé en mémoire, mais alors quand le décharger ? et bien c'est là qu'intervient le comptage de référence.

Son principe est très simple ... chaque fois que l'on obtient une référence sur un objet COM (celà inclu l'instanciation mais aussi l'obtention d'une nouvelle interface d'un objet déjà instancié) on incrémente un compteur interne au composant, et dès que l'on a plus besoin de l'objet on décrémente ce même compteur ! dès que le compteur tombe à zéro on décharge le composant de la mémoire.

Cette incrémentation/décrémentation est réalisée par les fonctions **Addref()** et **Release()** mais c'est au développeur d'implémenter ces deux fonctions et gérer le compteur en conséquence (heureusement pour vous des bibliothèques existent pour vous mâcher le travail). En effet en tant qu'heureux développeur COM/DCOM vous avez le devoir d'implémenter l'interface **IUnknown** <sup>7</sup> et donc de mettre du code derrière ces deux fonctions !

Voici un exemple d'implémentation de ces deux fonctions (le compteur de référence est un long : m\_cRef) :

---

<sup>6</sup> en COM seules les fonctions *Addref()* et *Release()* ne retournent pas de *HRESULT*.

<sup>7</sup> *AddRef()* et *Release()* ne retournent pas obligatoirement le nouveau compteur de référence, disons que ce n'est pas dans les spécifications de COM ... mais c'est mieux !



```
ULONG _stdcall CPhoneObject::AddRef(void)
{
    return InterlockedIncrement(&m_cRef);
}
```

```
ULONG _stdcall CPhoneObject::Release(void)
{
    if (0 == --m_cRef)
    {
        delete this;
        return 0;
    }
    return m_cRef;
}
```

### 3.1.2 Le changement d'interface

Nous avons vu qu'un objet COM peut posséder plusieurs interfaces ... et bien la question qui se pose ici c'est : quand on a un objet COM (instancié !) comment changer d'interface ? et bien c'est le rôle de la fonction **QueryInterface()**<sup>8</sup>.

Cette fonction prend un argument en entrée : l'IID (Interface Identifier) c'est en fait le GUID de l'interface désirée, et un argument en sortie qui est en fait le pointeur sur l'interface en retour.

Voici un exemple d'implémentation de cette fonction :

```
HRESULT _stdcall CPhoneObject::QueryInterface (REFIID riid, void **ppv)
{
    if (NULL == ppv)
        return E_INVALIDARG;
    *ppv = NULL;

    if (IID_IPhone == riid)
        *ppv = (IStream *) this;
    else
        if (IID_IUnknown == riid)
            *ppv = static_cast<IPhone *>( this );

    if (*ppv) {
        AddRef();
        return S_OK;
    }
    return E_NOINTERFACE;
}
```

---

<sup>8</sup> *QueryInterface()* fait un appel à *Addref()* !

## 3.2 L'interface IClassFactory

Maintenant que nous avons vu les fonctions de bases que doivent à tout pris implémenter les objets COM, nous allons nous pencher sur l'instantiation de ces derniers. Nous avons vu dans le premier chapitre (§ Les Composants) qu'un composant COM ne doit pas être instancié par une commande propre à un langage de programmation (comme l'opérateur new en C++).

En fait un composant COM doit s'instancier lui même, et retourner un pointeur sur l'objet obtenu ... Et bien ça c'est le boulot de la fabrique de Classe (ou Factory) ! Et dans le petit monde de COM c'est l'interface **IClassFactory** <sup>9</sup> qui est en charge de cette fonction (qui dit interface dit que c'est à vous d'implémenter le code qui va avec !!!).

Voici le prototype de IClassFactory:

```
struct IClassFactory : IUnknown
{
    virtual STDMETHODCALLTYPE CreateInstance(IUnknown* pUnkOuter,
                                             REFIID riid, void** ppv) = 0;
    virtual STDMETHODCALLTYPE LockServer(BOOL bLock) = 0;
};
```

### 3.2.1 Description des méthodes

La méthode qui correspond à l'instanciation d'un objet COM est **CreateInstance(...)**. Cette dernière prend trois arguments:

- **pUnkOuter** : qui sera utilisé dans le cas de l'aggrégation (Cf. § Héritage).
- **riid** : qui est le GUID de l'interface demandée sur le composant à instancier.
- **ppv**: qui est le pointeur retourné (sur l'interface du composant instancié).

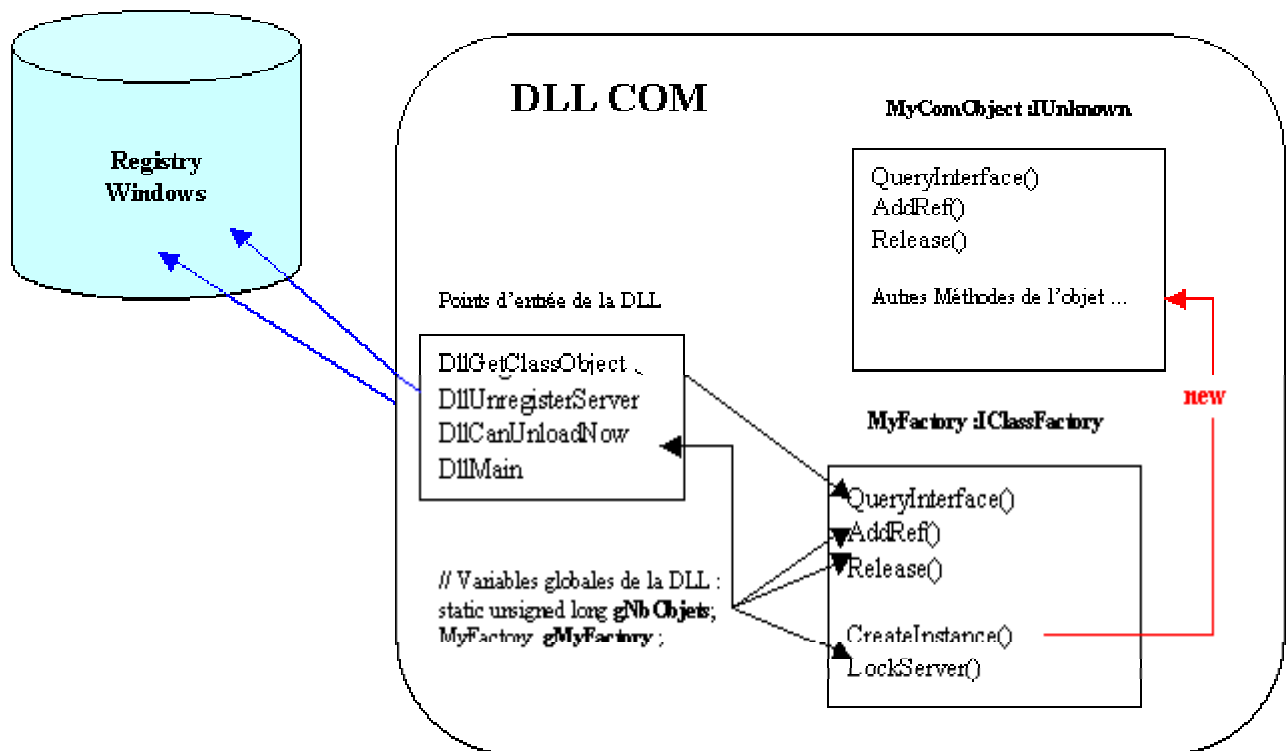
Quant à la méthode **LockServer(...)**, elle a pour rôle d'empêcher la destruction du composant ... En gros elle effectue simplement une incrémentation (ou une décrémentation selon la valeur de bLock) du compteur de référence de **IClassFactory**.

### 3.2.2 Synoptique de fonctionnement

Maintenant que nous avons vu les deux interfaces de base de tout composant COM, nous allons voir (dans le cas d'une DLL) comment elles interagissent entre elles et avec l'extérieur.

---

<sup>9</sup> cette interface (comme toute interface COM qui se respecte) dérive de IUnknown, et bien cela entraîne qu'elle va devoir elle aussi implémenter un comptage de référence et un changement d'interface !



Notons que la Factory est instanciée initialement en tant qu'objet global de la DLL. En fait cet objet va constituer la clé de voûte de l'objet COM. En effet dès qu'un client va demander un objet (COM !) via la fonction **CoGetClassObject()** COM va tout d'abord rechercher dans la registry la DLL (on est toujours dans le cas d'une DLL !) correspondant, la charger en mémoire (ce qui a pour conséquence d'instancier la Factory) puis faire un appel à **DllGetClassObject()**.

Cette fonction renvoie alors un pointeur sur la Factory (à condition d'avoir fourni l'IID de l'interface de la factory) ... il ne reste plus au client qu'à faire un appel à la méthode **CreateInstance()** du pointeur retourné pour avoir un nouveau pointeur vers l'objet désiré !

*Note: La fonction **CoCreateInstance()** effectue ce travail pour vous !*

Il est à noter d'autre part la présence d'un compteur de référence (**gNbObjets**) global à la DLL, c'est ce dernier qui effectue un comptage de référence pour la DLL (dès qu'il atteint 0, la DLL est déchargée).

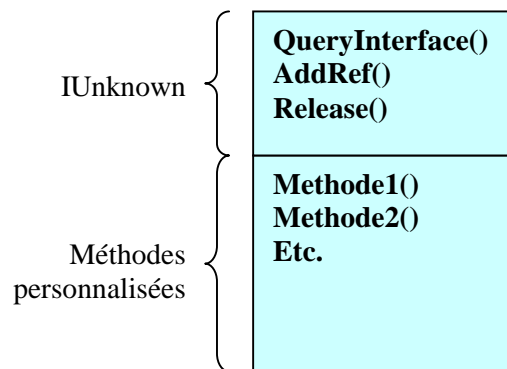
## 4 Automation

### 4.1 Introduction

Qu'est ce que Automation ? Et bien c'est en quelque sorte la possibilité que va nous offrir COM pour rendre accessible des composants COM à des langages qui ne connaissent pas les pointeurs de fonctions (comme Visual Basic par exemple).

→ C'est, pour résumé, le moyen fournit par COM pour permettre à deux applications de type complètement différent (VB, Script, C++, Etc.) de communiquer.

Bon c'est bien beau tout ça mais comment ça marche ? Et bien nous avons vu que chaque objet COM possède dans sa **VTABLE**<sup>10</sup> les méthodes de **IUnknown** puis ses propres méthodes :



Chaque méthode personnalisée correspondant à un nouveau pointeur de fonction. Et Bien Automation va définir un nouveau mode d'accès aux méthodes de l'objet COM.

### 4.2 IDispatch

Chaque appel de méthode personnalisée passera par une méthode unique. Par exemple au lieu de dire « j'appelle la méthode Tartenpion() de mon objet COM » on dira « j'appelle la méthode **Invoke()** de mon objet COM qui, elle, fera l'appel à ma fonction Tartenpion() ».

Pour résumer on crée par ce biais une nouvelle indirection ... Grâce à elle on rend complètement invisible vis à vis de l'extérieur la description de l'objet COM !

Cette opération est apportée par l'apport d'une nouvelle interface à implémentée (...Et oui, c'est encore au développeur de faire tout le boulot ... mais ne l'oublions pas, COM n'est qu'un protocole logiciel !) **IDispatch** qui dérive naturellement de **IUnknown** :

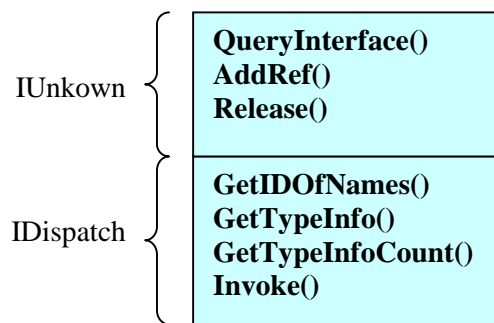
<sup>10</sup> Je rappelle qu'une VTABLE est en fait la table de pointeurs de fonctions correspondant à une classe virtuelle.



Voici son prototype (en C++ naturellement !) :

```
class IDispatch : IUnknown
{
public:
    virtual HRESULT GetIDsOfNames(REFIID riid, OLECHAR FAR* FAR* rgpszNames,
                                  unsigned int cNames, LCID lcid,
                                  DISPID FAR* rgDispId ) = 0;
    virtual HRESULT GetTypeInfo(unsigned int iTInfo, LCID lcid,
                                 ITypeInfo FAR* FAR* ppTInfo ) = 0;
    Virtual HRESULT GetTypeInfoCount(unsigned int FAR* pctinfo ) = 0;
    Virtual HRESULT Invoke(DISPID dispIdMember, REFIID riid, LCID lcid,
                           WORD wFlags,
                           DISPPARAMS FAR* pDispParams,
                           VARIANT FAR* pVarResult,
                           EXCEPINFO FAR* pExcepInfo,
                           unsigned int FAR* puArgErr ) = 0;
};
```

Ainsi que sa VTABLE :



C'est bien tout ça mais ça cache beaucoup de problème sous-jacents, comme le mécanisme de passage arguments.

→ Et oui il va falloir définir Comment spécifier:

- Le nombre d'argument à passer.
- Leur dénomination.
- Leurs type.
- Le retour éventuel de fonctions.

Et bien l'interface **IDispatch** répond à tout cela !

En fait tout se passe comme si au lieu d'être identifiée par un nom, une méthode est identifiée par un numéro : le **DISPID**. Avant tout appel de méthode, le développeur doit trouver le DISPID correspondant à la sa méthode puis le passer à une autre méthode de IDispatch pour exécuter la méthode demandée.

Et bien c'est le rôle de la méthode **GetIDOfNames(...)** qui effectue la correspondance : DISPID ↔ « Nom de méthode » quant à **Invoke(...)** elle effectue l'appel à la fonction proprement dite.

Le deux autres fonctions **GetTypeInfoCount(...)** et **GetTypeInfo(...)** elles sont en charges de retourner des informations de type pour l'objet.



## 4.3 Le type Variant

Afin de fonctionner correctement Automation impose l'utilisation d'un type standard : sorte de type que regroupe tous les types les plus utilisés et qui impose leur représentation mémoire (qui dépend normalement de la machine, OS, Etc.). Ce type est le **VARIANT** et en fait ce n'est qu'une structure composée d'une énumération C comme suit :

```
typedef struct tagVARIANT {
    VARTYPE vt;
    unsigned short wReserved1;
    unsigned short wReserved2;
    unsigned short wReserved3;
    union {
        unsigned char bVal;
        short iVal;
        long lVal;
        float fltVal;
        double dblVal;
        VARIANT_BOOL boolVal;
        SCODE scode;
        CY cyVal;
        DATE date;
        BSTR bstrVal;
        IUnknown FAR* punkVal;
        IDispatch FAR* pdispVal;
        SAFEARRAY FAR* parray;
        unsigned char FAR* pbVal;
        short FAR* piVal;
        long FAR* plVal;
        float FAR* plfltVal;
        double FAR* pdblVal;
        VARIANT_BOOL FAR* pboolVal;
        SCODE FAR* pscode;
        CY FAR* pcyVal;
        DATE FAR* pdate;
        BSTR FAR* pbstrVal;
        IUnknown FAR* FAR* ppunkVal;
        IDispatch FAR* FAR* ppdispVal;
        SAFEARRAY FAR* FAR* pparray;
        VARIANT FAR* pvarVal;
        void FAR* byref;
    };
};
```

➔ Tous les types qui figurent dans l'énumération (double, float, BSTR, Etc.) sont définis comme étant les type compatibles Automations.

Le fonctionnement est très simple puisque la variable (de la structure) **vt** désigne de quel type le variant est composé. Par exemple si **vt = VT\_BSTR** l'énumération contiendra une donnée de type BSTR (chaîne de caractère) et il faudra donc prendre le membre **bstrVal**.

Voici un tableau de correspondance des valeurs de vt avec leur type correspondant :





*bVal* → VT\_UI1.  
*iVal* → VT\_I2.  
*lVal* → VT\_I4.  
*fltVal* → VT\_R4.  
*dblVal* → VT\_R8.  
*boolVal* → VT\_BOOL.  
*scode* → VT\_ERROR.  
*cyVal* → VT\_CY.  
*date* → VT\_DATE.  
*bstrVal* → VT\_BSTR.  
*punkVal* → VT\_UNKNOWN.  
*pdispVal* → VT\_DISPATCH.  
*parray* → VT\_ARRAY | \*.  
*pbVal* → VT\_BYREF | VT\_UI1.  
*piVal* → VT\_BYREF | VT\_I2.  
*plVal* → VT\_BYREF | VT\_I4.  
*pfltVal* → VT\_BYREF | VT\_R4.  
*pdblVal* → VT\_BYREF | VT\_R8.  
*pboolVal* → VT\_BYREF | VT\_BOOL.  
*pcode* → VT\_BYREF | VT\_ERROR.  
*pcyVal* → VT\_BYREF | VT\_CY.  
*pdate* → VT\_BYREF | VT\_DATE.  
*pbstrVal* → VT\_BYREF | VT\_BSTR.  
*ppunkVal*; → VT\_BYREF | VT\_UNKNOWN.  
*ppdispVal*; → VT\_BYREF | VT\_DISPATCH.  
*pparray*; → VT\_ARRAY | \*.  
*pvarVal* → VT\_BYREF | VT\_VARIANT.

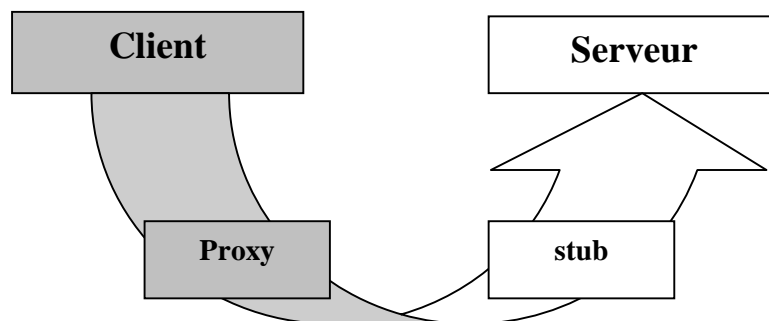
*Note : les structures sont interdites en Automation.*

## 5 Les types d'interfaces COM

### 5.1 Le marshalling

Avant toute chose il serait important de définir (ou plutôt d'expliquer) ce qu'est le marshalling ? Et bien COM devant être indépendant du type de machine (et donc de la représentation interne des données) ... il doit fournir un mécanisme de transformation de données ! c'est l'un des rôles du marshalling.

Plus complexe, vous êtes dans un mode de communication inter-processus (par exemple dans votre application vous avez instancié un objet COM EXE, ou alors vous instanciez via DCOM une DLL sur une autre machine !) ... comment allez vous faire pour passer vos pointeurs (arguments), qui n'ont de sens que dans votre espace mémoire, à l'autre processus ? Et bien voilà, c'est là le rôle principal du marshalling !



Le schéma ci-avant montre le mécanisme de marshalling ... Ce dernier fait intervenir deux modules (qui sont en fait le code de marshalling compilé en DLL) **proxy** et **stub**. Le premier est chargé de passer les arguments au stub qui les passe (au bon format) au serveur.

### 5.2 dispInterface

Automation pourrait se suffire à lui même puisque nous pouvons appeler toutes les méthodes de notre objet, de plus il permet de casser l'immuabilité des interfaces ; en effet, la VTABLE de IDispatch (ou **dispinterface**) ne changera jamais, même si on ajoute/modifie/supprime des méthodes à notre objet ... mais bon, ça reste un mécanisme complexe, et surtout très peu performant !

*Important : Microsoft fournit le code de Marshaling pour Automation (le FreeThreadedMarshaller).*

### 5.3 Custom Interface

En fait, jusqu'à Automation les interface que nous avons vu étaient de type personnalisées (**Custom Interface**). C'est à dire qu'elles étaient amenées à changer dès que l'on opérait une modification (ajout de méthode, changement de type d'argument, Etc.). Ces interfaces sont très performantes du fait que la VTABLE permet un accès direct à la méthode.

Cependant elles impliquent leur immuabilité , c'est à dire que l'on ne doit y apporter aucunes modifications une fois qu'elles sont mises en production.

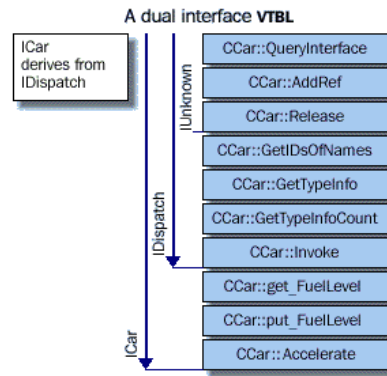


D'autre part, les types de données mis en œuvres étant libres (ils peuvent être différents que ceux contenus dans un Variant puisque qu'une interface personnalisée n'est pas compatible Automation) le développeur de l'objet COM doit fournir le code de marshallng !

## 5.4 Interfaces duales

Une interface duale pourrait être résumée par l'équation :

**Custom Interface + dispInterface = dual Interface**



Le principe est d'utiliser l'interface IDispatch pour les clients Automation, mais de laisser l'interface personnalisée pour un accès plus rapide (développeurs C++).