

Ambientes de Desenvolvimento Avançados
Engenharia Informática
Instituto Superior de Engenharia do Porto

Teófilo Matos
2001

ÍNDICE

INTRODUÇÃO	5
O VOCABULÁRIO COM	7
UM OBJECTO COM TÍPICO	8
O INTERFACE	9
MÉTODOS DO IUNKNOWN	12
MÉTODOS DO IDISPATCH	12
O GUID	13
INTERACÇÃO ENTRE O CLIENTE E O SERVIDOR COM	15
O SERVIDOR COM	17
CONSTRUÇÃO DE UM SERVIDOR COM <i>IN-PROCESS</i> (DLL)	20
CRIAR UM SERVIDOR COM UTILIZANDO O WIZARD-ATL	21
ADICIONAR UM OBJECTO COM	25
ADICIONAR UM MÉTODO AO SERVIDOR RECORRENDO AO WIZARD	29
ADICIONAR UM MÉTODO AO SERVIDOR SEM RECORRER AO WIZARD	32
FICHEIROS RESULTANTES	34
PASSOS NECESSÁRIOS À DISPONIBILIZAÇÃO DO SERVIDOR:	36
REGISTO DO SERVIDOR	37
O CLIENTE COM	38
CLIENTE COM EM VISUAL BASIC	39
CLIENTE COM EM VISUAL C++	42
FAZER COM QUE A APLICAÇÃO SEJA UM CLIENTE COM	49
DEFINIR MÉTODOS CLIENTES QUE INVOQUEM OS MÉTODOS DO SERVIDOR COM	50
UTILIZAÇÃO DE COMPONENTES DE TERCEIROS	53
CLIENTE EM VISUAL BASIC	54
CLIENTE EM VISUAL C++	57
EXERCÍCIO DE REFERÊNCIA – FASE 3.	61
CRIAR O SERVIDOR UTILIZANDO O WIZARD-ATL	62
ADICIONAR O OBJECTO COM PRODUTO	65
ADICIONAR O MÉTODO <i>TEMStockSuficiente</i> AO SERVIDOR	68
ADICIONAR O CÓDIGO AO MÉTODO <i>TEMStockSuficiente</i>	73
COMO INVOCAR O MÉTODO A PARTIR DA NOSSA APLICAÇÃO CLIENTE:	75
CONCLUSÃO	77
ANEXOS	78

Introdução

Com este documento, pretende-se fazer uma introdução ao COM e ilustrar, recorrendo a tutoriais, as partes essenciais da construção de servidores e clientes COM.

Este documento não tem como objectivo principal abordar de forma exaustiva todas as características e funcionalidades do COM, uma vez que este tópico é um campo muito vasto o que envolveria uma descrição muito mais detalhada dos pormenores inerentes ao subsistema COM.

à

O COM é um sistema independente da plataforma de utilização, que permite desenvolver objectos (componentes binários) que podem interagir com outros objectos. Podendo esses objectos estar a correr no mesmo processo, num outro processo, e, até, numa máquina remota. Esses objectos podem até ter sido desenvolvidos por linguagens diferentes em sistemas operativos diferentes. Portanto, o COM, deve ser entendido como um *standard binário* e não como uma linguagem de programação orientada a objectos. Estes objectos podem ser distribuídos na forma de componentes binários e serem alterados ou modificados sem comprometerem os clientes antigos. Estes objectos, podem também, ser mudados de local na rede de forma transparente para os seus clientes.

Esta tecnologia disponibiliza o suporte ideal para desenvolver aplicação robustas, portáteis e escalonáveis.

Estes objectos podem ser criados por uma grande variedade de linguagens de programação, sendo o Microsoft Visual C++ uma das eleitas pelo facto de disponibilizar mecanismos que simplificam a implementação dos mesmos.

Os objectos COM podem ser entendidos, como disponibilizando: quer um conjunto de dados, quer um conjunto de funções relacionadas de acesso e manipulação desses mesmos dados. A estes conjuntos de funções chamam-se *Interfaces*, e às funções pertencentes ao *interface* chamam-se métodos. O COM obriga a que para se aceder a métodos de um determinado *interface*, seja necessário a obtenção de um apontador para o referido *interface*.

Como exemplos da utilização do COM (como tecnologia de raiz) temos o *OLE* e o *ActiveX*.

Um outro requisito do COM prende-se com o facto de este obrigar a que cada objecto seja único em todo o mundo. Entenda-se aqui que um objecto COM significa um componente binário, pertencente a um servidor COM, e que possui uma identificação única. Este parece ser, à primeira vista, um requisito um pouco exagerado, dado o número elevado de objectos existentes na rede mundial. Porém, se considerarmos a Internet como sendo uma rede mundial, esta exigência faz todo o sentido. Este problema é ultrapassado através de um conceito denominado de GUID - *globally unique identifier* – [ver **GUID**], ou seja, um identificador único universal.

Como já foi mencionado, os objectos COM têm a particularidade de poderem ser escritos numa linguagem e num sistema operativo completamente diferentes. Para além disso, têm a capacidade de ser remotamente acedidos, ou seja, podem estar a correr numa *thread*, processo ou computador diferentes, podendo este computador estar ainda a correr um sistema operativo diferente. Por isso, e tendo em conta os aspectos atrás referidos, existe a necessidade de transmitir parâmetros através da rede a objectos de outras máquinas. Este problema é resolvido através da criação de uma nova forma de especificar o interface entre o cliente e o servidor. Existe um novo compilador chamado IDL –*Interface Definition Language*– [ver **MIDL**] que permite fazer essa especificação. Este define objectos COM, interfaces, métodos e parâmetros.

O vocabulário COM

Partindo do pressuposto que é conhecido o vocabulário da linguagem de programação C++, irá ser apresentada, na tabela seguinte, uma comparação entre a terminologia convencional e a que é usada no COM.

Conceito	Convencional (C++/OO)	COM
Cliente	Um programa que requisita um serviço de um servidor.	Um programa que chama métodos COM.
Servidor	Um programa que fornece serviços a outros programas.	Um programa que torna os objectos COM disponíveis a clientes COM.
Interface	nenhum	Um apontador para um grupo de funções que são chamadas através do COM.
Classe	Um tipo de dados. Define um grupo de métodos e dados que são utilizados em conjunto.	A definição de um objecto que implementa um ou mais interfaces COM. Também conhecido como coclass.
Objecto	Uma instância de uma classe.	Uma instância de uma coclass,
<i>Marshalling</i>	nenhum	Transferência de dados entre o cliente e o servidor.

Um objecto COM típico

Supondo que se pretende criar um objecto COM simples que suporte apenas um interface, que, por sua vez, contenha um único método, eis os passos que deveriam ser seguidos:

1	Criar um objecto COM e dar-lhe um nome. Este objecto será implementado dentro de um servidor COM.
2	Definir o interface e dar-lhe um nome.
3	Definir a função no interface e dar-lhe um nome.
4	Instalar o servidor COM.

Um problema que se põe de imediato tem a ver com o nome dos objectos. Admitindo que:

- todas as máquinas no universo COM têm a possibilidade de suportar múltiplos servidores COM;
- cada um destes servidores contém um ou mais objectos COM;
- cada um dos objectos, atrás referidos, implemente um ou mais interfaces;
- estes servidores possam ser criados por uma variedade de programadores, não existindo nada que os impeça de escolherem nomes idênticos;
- da mesma forma, os objectos COM expõem um ou mais nomes de interfaces, também criados por múltiplos programadores que podem aleatoriamente escolher nomes iguais;

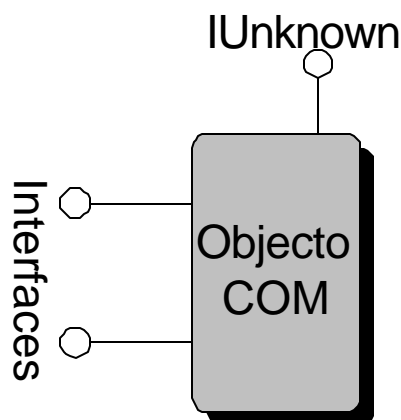
algo tem de ser feito para prevenir a colisão de nomes, caso contrário, gera-se uma grande confusão. Como já foi referido inicialmente, o conceito GUID [**ver GUID**] resolve o problema de “Como conseguir manter todos estes nomes únicos”.

O Interface

De uma forma simples, pode-se dizer que um interface não é mais do que uma colecção de funções. O COM permite que a coclass (*COM class* ou *Component Object Class*) tenha múltiplos interfaces, tendo cada um o seu próprio nome e a sua própria colecção de funções.

Uma das regras do COM é que apenas se pode aceder aos objectos através do seu interface. O programa cliente está completamente isolado da implementação do servidor através dos interfaces.

O programa cliente não sabe nada acerca do objecto COM ou da classe C++ que o objecto COM implementa. O cliente apenas “vê” as partes do objecto que lhe são expostas.

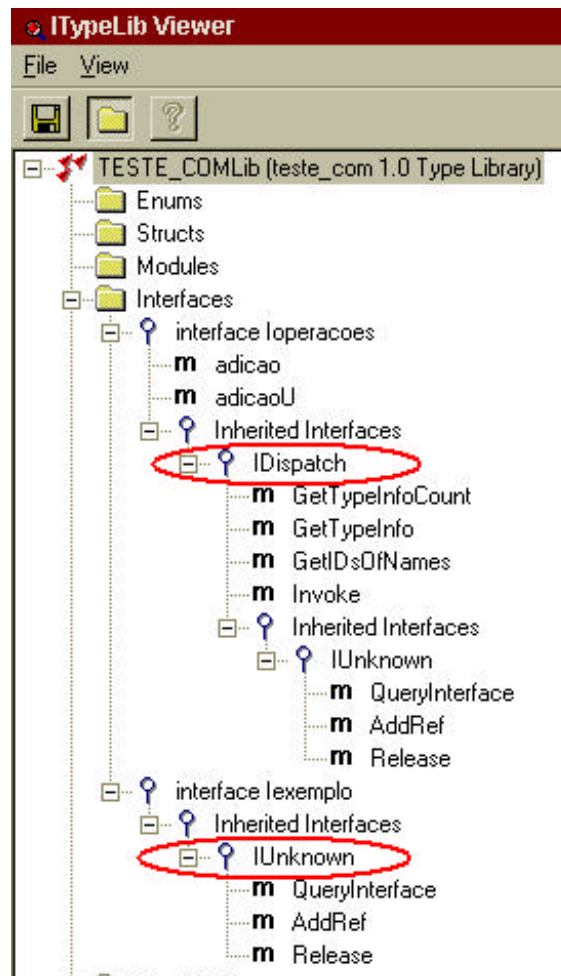


O isolamento dos métodos dos interfaces e da implementação é crucial para o COM. Ao isolar o interface da sua implementação, é possível construir componentes que podem ser re-usados e, por sua vez, alterados, visto a sua implementação e localização ser totalmente transparente para os clientes. Esta característica simplifica e multiplica a utilidade de um objecto.

Um facto importante a ter em conta é que o nome de um interface COM é único. Isto é, o programador pode partir do pressuposto que, no COM, se acede a um interface com um determinado nome. As funções membro do interface e os seus parâmetros serão exactamente os mesmos em todos os objectos COM que implementem o interface. Se for pretendido alterar a função membro de um interface, terá de ser criado um interface com um novo nome.

O interface *IUnknown* é a base de todos os interfaces COM. Normalmente, a implementação deste interface é escondida por níveis elevados de abstracção, que são utilizados para construir os objectos COM. O *IUnknown* é semelhante a uma classe abstracta base da linguagem C++. Todos os interfaces COM devem derivar do interface *IUnknown*. Este interface trata da criação e manutenção dos interfaces. Os métodos do *IUnknown* são usados para criar – *QueryInterface* - , contar referências – *AddRef* - e libertar um objecto – *Release*. Todos os interfaces implementam estes três métodos, que são usados internamente pelo COM.

Através do utilitário *OLE/COM Object Viewer* podemos verificar quais as características de um determinado Servidor COM. Por exemplo, podemos verificar que o servidor *TesteCOM.dll* disponibiliza o interface **loperacoes** e **lexemplo**, e que o interface *loperacoes* possui os métodos *adicao* e *adicaoU*. Também se pode verificar que o interface *loperacoes* deriva do interface *IDispatch* que por sua vez deriva do interface *IUnknown*. Enquanto o interface *lexemplo* deriva directamente do *IUnknown*.



Todos os interfaces do COM são derivados, directa ou indirectamente do interface *IUnknown*.

O interface *IDispatch* permite que os objectos, métodos e propriedades fiquem disponíveis para ferramentas de programação e aplicações que suportem *Automation* [ver **Automation**]. O COM, ao permitir que os objectos implementem este interface, fornece a clientes *Automation* acesso aos objectos. Exemplos de clientes *Automation* são: Visual Basic, VB Script, etc.

Métodos do IUnknown

QueryInterface	Retorna apontadores para interfaces.
AddRef	Incrementa o contador de referências.
Release	Decrementa o contador de referências.

Métodos do IDispatch

GetTypeInfoCount	Verifica se o objecto disponibiliza informação em <i>runtime</i> (0 – não; 1 – sim). Esta informação pode ser útil para Browsers, compiladores, etc.
GetTypeInfo	Obtém informação sobre o objecto. Esta informação pode ser diferente em função da linguagem.
GetIDsOfNames	Permite obter um identificador para um método ou propriedade para posteriormente ser utilizada pelo <i>Invoke</i> .
Invoke	Permite aceder a métodos e propriedades disponibilizadas pelo objecto.

O GUID

A OSF (*Open Software Foundation*) desenvolveu um algoritmo que combina o endereço da rede, a data (em incrementos de 10 nanosegundos), e um contador. O resultado é um número de 128-bits que é único.

O número 2^{128} é um número extremamente extenso. É possível identificar cada nanosegundo desde o início do Universo – restando ainda 39-bits. A OSF chama a este número o UUID (*Universal Unique Identifier*), ou seja, o identificador único universal. A Microsoft utiliza o mesmo algoritmo para o standard de nomes para o COM. No COM, a Microsoft decidiu rebatizá-lo de GUID (*Global Unique Identifier*), ou seja, um identificador único global.

A convenção para a escrita de GUID's é em hexadecimal. Um GUID apresenta-se, tipicamente, da seguinte forma:

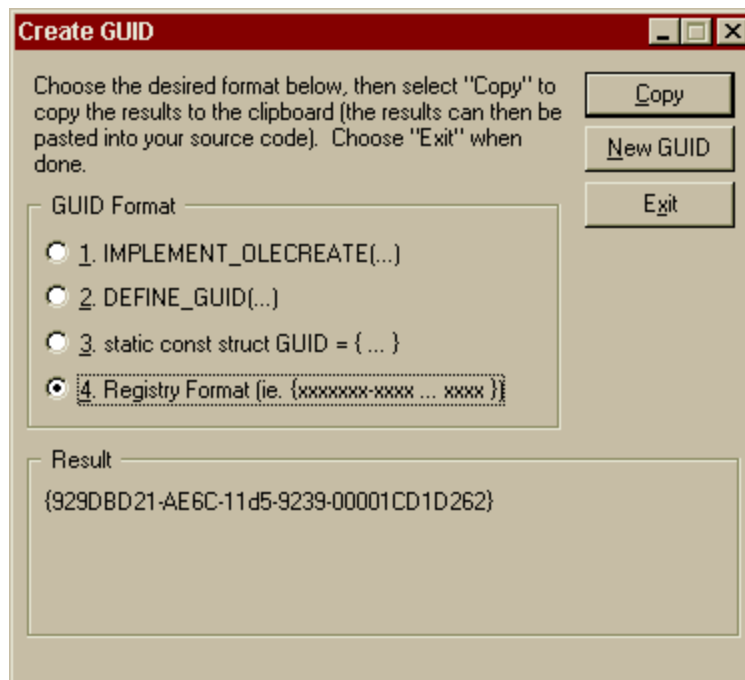
```
{929DBD21-AE6C-11d5-9239-00001CD1D262}
```

Como não existe um tipo de dados em C++ de 128-bits, é usada uma estrutura para a sua representação. Essa estrutura consiste em quatro campos diferentes. Normalmente, nunca será necessário manipular cada um dos campos individualmente. A estrutura é sempre utilizada inteira.

```
typedef struct _GUID
{
    unsigned long Data1;        // 4 bytes +
    unsigned short Data2;      // 2 bytes +
    unsigned short Data3;      // 2 bytes +
    unsigned char Data4[8];     // 8 bytes = 16 bytes = 128 bits
} GUID;
```

Estrutura do GUID

Os GUID's podem ser gerados por programas chamados de GUIDGEN. Nestes programas, apenas temos que pedir um novo número para que seja gerado um novo GUID. Cada GUID que é gerado, dá-nos a garantia de ser único, não interessando a quantidade de GUID's que se tenha gerado até ao momento e nem de quantas pessoas no planeta que os tenham gerado. Esta unicidade existe porque todas as máquinas têm, por definição, um endereço único. De qualquer forma, a máquina deve ter um endereço de rede para que os programas GUIDGEN funcionem no seu potencial máximo. Mesmo que a máquina não tenha um endereço de rede, o GUIDGEN irá inventar um, mas a probabilidade de unicidade fica diminuída.



Gerador de GUID's

Tanto os objectos COM como os interfaces COM têm GUID's para a sua identificação.

Interacção entre o Cliente e o Servidor COM

No COM, os programas cliente tratam de tudo, sendo os servidores totalmente passivos, pois apenas respondem a pedidos. O cliente COM é, então, um conjunto de código ou um objecto que obtém um apontador para um servidor COM e, através desse apontador, acede aos seus serviços chamando métodos nos seus interfaces.

Isto significa que os servidores COM comportam-se de uma maneira síncrona através de chamadas individuais a métodos por parte do cliente.

Regra geral, a afirmação atrás feita é correcta, exceptuando-se os casos em que o servidor COM implementa *Connection points*. Na prática, esses 'pontos de ligação', o que fazem é permitirem que o servidor implemente interfaces de um objecto COM seu cliente, dando-lhe a possibilidade de despoletar métodos nesse cliente por sua própria iniciativa. Esta situação pode tornar-se útil, no sentido em que possibilita ao servidor notificar o(s) cliente(s) a ele ligado(s) e que implementem o interface por ele conhecido.

Na tabela seguinte é mostrado o processo de interacção entre uma aplicação cliente e um servidor COM:

Pedido do Cliente	Resposta do Servidor
Pede acesso a um interface COM, especificando a classe e o interface (através do GUID)	<ol style="list-style-type: none">1. Inicia o servidor (se necessário). Se este é um servidor <i>In-Process</i>, a DLL é lida para memória. Servidores EXE serão lidos pelo SCM.2. Cria o objecto COM pedido.3. Cria o interface para o objecto COM.4. Incrementa o contador de referências do interface activo.5. Retorna o interface para o cliente.
Invoca um método do interface	Executa o método no objecto COM.
Liberta o interface	<ol style="list-style-type: none">1. Decrementa o contador de referências do interface.2. Se o contador de referências é igual a zero, o objecto COM é apagado.3. Se não existirem mais conexões activas, o servidor é desligado. Alguns servidores não se desligam por eles próprios.

Interacção entre o Cliente e o Servidor COM

A melhor forma de compreender o COM, é olharmos para ele da perspectiva de uma aplicação cliente. O objectivo da programação COM é desenvolver objectos com utilidade, para que sejam disponibilizados a aplicações clientes COM. Um cliente COM é um programa que utiliza o COM para chamar métodos de um servidor COM.

O Servidor COM

O servidor COM é um qualquer objecto que disponibiliza serviços a clientes, estando estes serviços na forma de interfaces COM, podendo ser chamados por qualquer cliente que seja capaz de obter um apontador para os interfaces do servidor COM.

Os servidores COM apresentam-se em três tipos:

<i>In-process</i>
<i>out-of-process</i>
Serviços do Windows NT

Os objectos COM são idênticos, independentemente do tipo do servidor utilizado. Os interfaces COM e as coclass (classes COM) não se importam com o tipo de servidor que está a ser utilizado. Para o programa cliente, o tipo de servidor é quase totalmente transparente. Contudo, escrever estes servidores pode ser significativamente diferente para cada um dos tipos.

- Servidores *In-process* são implementados como *Dynamic Link Libraries* (DLL's). Isto significa que o servidor é dinamicamente carregado no processo cliente em *run-time*. O servidor COM passa a ser parte da aplicação cliente e as operações com o COM são feitas através de *threads*. Tradicionalmente, esta é a forma como os objectos COM são implementados porque a performance é realmente elevada e o *overhead* é mínimo para uma chamada a uma função COM. O subsistema COM trata automaticamente de carregar e descarregar a DLL da memória.

- Servidores *out-of-process* têm uma linha mais clara de separação entre o cliente e o servidor. Estes tipos de servidores correm como programas executáveis (EXE) separados, e por isso num espaço e processo privado. O início e o fim do servidor EXE é tratado pelo SCM (*Service Control Manager*) do Windows. As chamadas a interfaces COM são tratadas através de mecanismos de comunicação entre processos. O servidor pode estar a correr no computador local ou num computador remoto. Se o servidor estiver num computador remoto, estamos falar de COM distribuído, ou seja, o DCOM.
- Servidores como 'serviços do Windows NT'. Um serviço é um programa que é automaticamente tratado pelo Windows NT, e não está associado ao utilizador. Isto significa que estes serviços podem arrancar automaticamente quando a máquina é iniciada e conseguem correr mesmo que ninguém esteja ligado ao Windows NT. Os serviços são uma maneira excelente de correr aplicações COM servidoras.
- Existe um quarto tipo de servidor, chamado de *surrogate*. Este é essencialmente um programa que permite a um servidor *In-process* correr remotamente. Estes servidores são úteis porque possibilitam a servidores baseados em DLL's ficarem disponíveis através de uma rede.

Para o cliente, o mecanismo de programação COM é bastante simples. A aplicação cliente pede ao subsistema COM por um componente em particular e este quase que por magia, é disponibilizado ao cliente.

Devido ao peso e complexidade do desenvolvimento OLE-COM, a Microsoft criou uma nova ferramenta chamada ATL (*Active Template Library*). Para os programadores de COM, o ATL é sem sombra de dúvida a ferramenta actualmente mais prática de ser utilizada. Através da utilização do *Wizard* ATL, a tarefa de construção de servidores COM torna-se relativamente simples se não quisermos descer a um nível muito baixo de programação.

O servidor COM que irá ser criado a título de exemplo, recorre ao *Wizard ATL*.

Os servidores ATL não se parecem com programas tradicionais. Um servidor COM, é na realidade uma colaboração de muitos e diversos componentes:

A aplicação
O subsistema COM
As classes ATL Template
O código IDL e a geração dos <i>headers</i> em C e programas, pelo MIDL
O registo do sistema (<i>registry</i>)

Pode ser difícil “olhar” para uma aplicação COM baseada em ATL tentando vê-la como um conjunto compacto. Mesmo sabendo o que estamos a fazer, existem muitas características da aplicação que não conseguimos ver. Grande parte da lógica do servidor está camuflada nos ficheiros de *header* do ATL. Não é possível encontrar nenhuma função *main()* que controle o servidor, mas apenas encontramos uma *shell* que permite fazer chamadas aos objectos ATL.

De seguida, serão agregadas todas as partes necessárias para fazer com que o servidor execute. Será, inicialmente, criado um servidor, utilizando-se, para tal, o *Wizard ATL COM*. Posteriormente, será adicionado ao servidor um objecto e um método. O servidor a ser criado será um *In-process*, visto ser o de mais fácil implementação, e evitando também a necessidade de termos de criar os objectos *proxy* e *stub*.

Construção de um Servidor COM *In-Process* (DLL)

Um servidor *In-Process* é uma biblioteca COM que é carregada com o programa em *run-time*. Por outras palavras, é um objecto COM dentro de uma DLL (*Dynamic Link Library*). Uma DLL não é um servidor no verdadeiro sentido da palavra porque ele é carregado no espaço de endereçamento do cliente.

Sempre que um cliente precisa de um objecto, o COM localiza o servidor (DLL) e automaticamente faz o seu carregamento. Uma vez este carregado, a DLL possui uma fábrica de classes (*class factory*) para a criação do objecto COM requisitado.

Sempre que a DLL não é necessária, esta é descarregada da memória. Esta situação pode ocorrer quando é feito o *Release* do objecto ou o *CoUninitialize*.

Existem vantagens e desvantagens na utilização de servidores COM *In-process*. A parte referente ao carregamento, libertação e exportação de funções da DLL é totalmente tratada pelo COM. Todo este processo adiciona muito pouco *overhead* ao programa. Com os servidores ligados a clientes perdem-se muitos aspectos importantes de distribuição de funcionalidades (DCOM).

Criar um Servidor COM utilizando o *Wizard-ATL*

Para que seja mais facilmente compreendido o processo de criação de um servidor COM, irá ser construído um passo-a-passo, utilizando-se para o efeito o *Wizard-ATL* do Visual Studio 6.0.

O servidor será muito simples e a sua finalidade é a de demonstrar a forma como pode ser utilizado para a construção do esqueleto do servidor, e alertar para as componentes essenciais dos servidores COM.

Este, possuirá, inicialmente, apenas um método que permitirá fazer a soma de dois valores inteiros longos – chamado de ***Adicao***.

O *Wizard ATL* permite que sejam seleccionadas todas as operações básicas para o servidor e fazer a geração da maior parte do código necessário.

No processo a seguir apresentado será criado um servidor chamado TesteCOM. Todos os servidores COM têm de possuir pelo menos um interface. O interface a ser criado para este servidor terá o nome IOperacoes. Os interfaces podem ter praticamente todos os nomes possíveis, mas, deve-se colocar sempre um prefixo 'I', se queremos seguir a convenção standard de nomes.

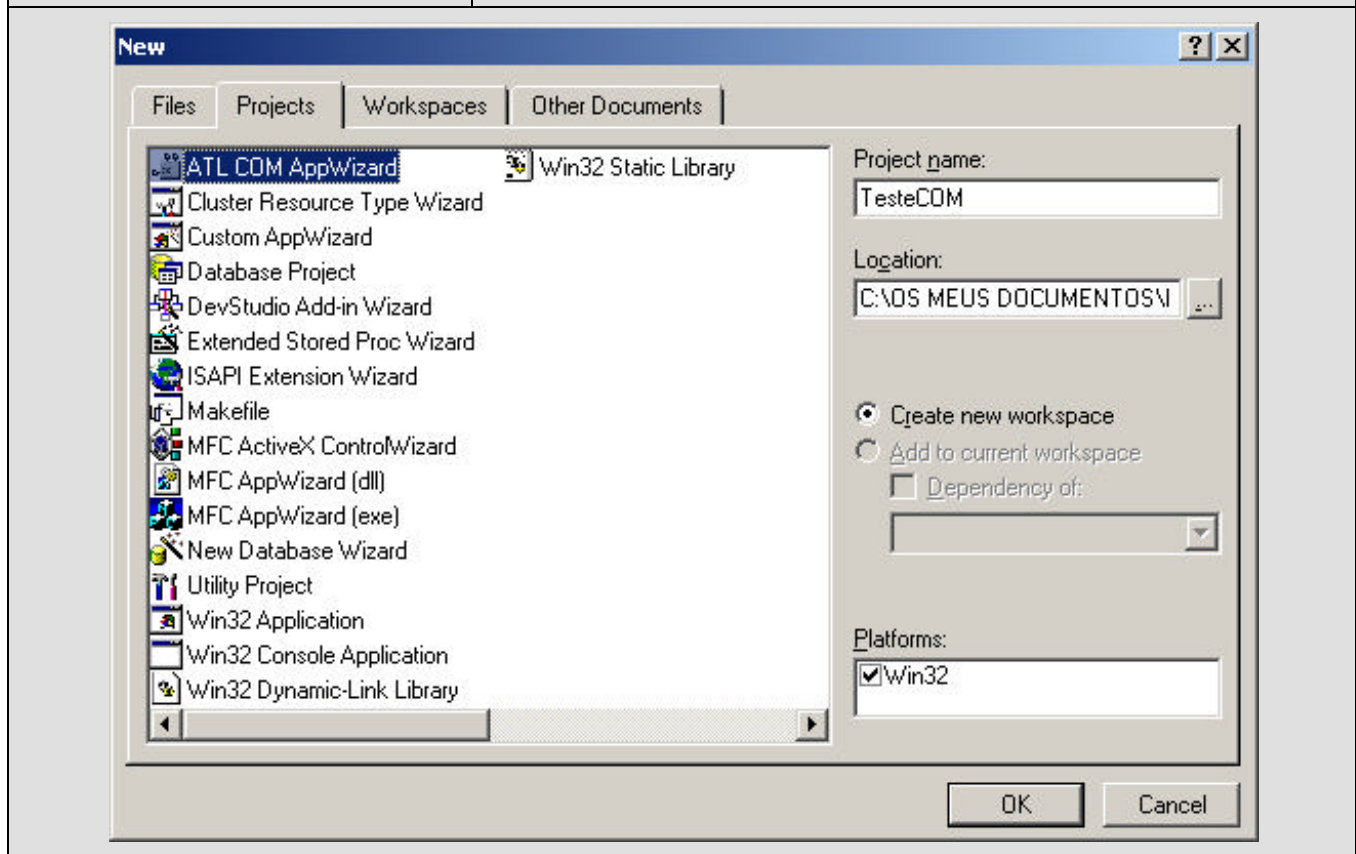
Passo 1

Primeiro seleccionamos a opção File|New... do menu principal.

Passo 2

De seguida seleccionamos a Tab *Projects* da janela *New*, e escolhemos **ATL COM AppWizard** da lista de projectos disponível e digitamos o seguinte:

Project Name	TesteCOM
Location	A que pretendemos para o nosso projecto
<input checked="" type="radio"/> Create New Workspace	

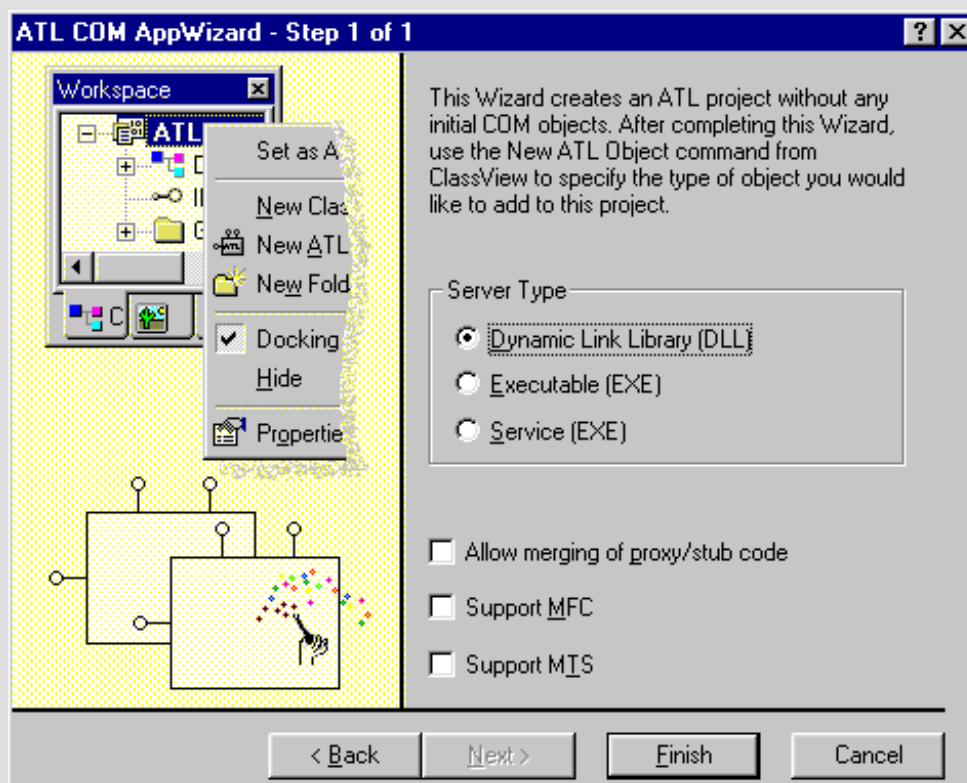


Passo 3

Depois, na janela seguinte, devemos seleccionar apenas que pretendemos criar um servidor *In-process* (dll):

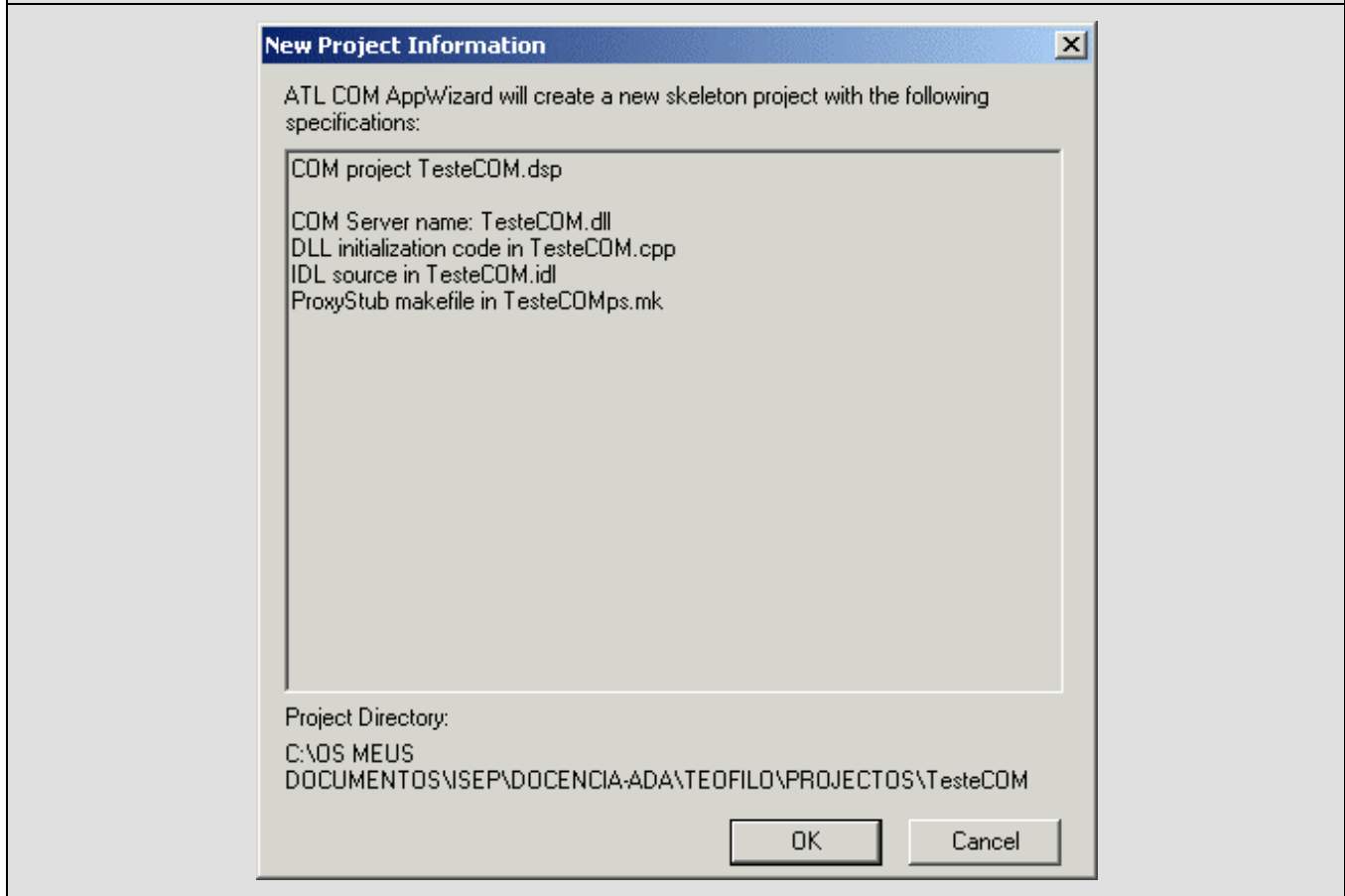
Server Type

Dynamic Link Library (DLL)



Passo 4

Finalizar o processo, carregando no botão *Finish*.

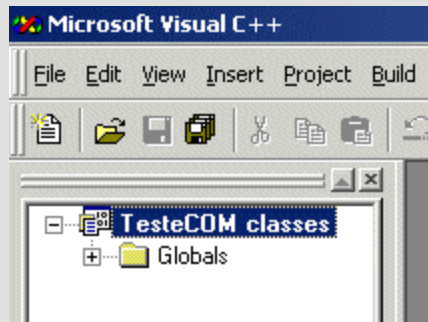


Após ter realizado estes passos, o *Wizard ATL* criará um projecto com todos os ficheiros necessários ao Servidor COM *In-process* - baseado em DLL. Embora este servidor compile e execute, não irá fazer nada. Para que possa ter utilidade, é preciso criar um interface e uma classe que suporte esse interface. Para além disso, deverão ser criados métodos nesse interface.

Adicionar um objecto COM

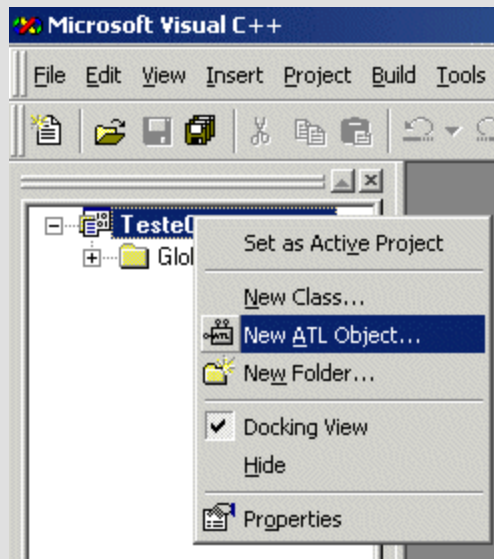
Passo 1

No *ClassView*, carregar no botão do lado direito do rato sobre o item “TesteCOM classes”.



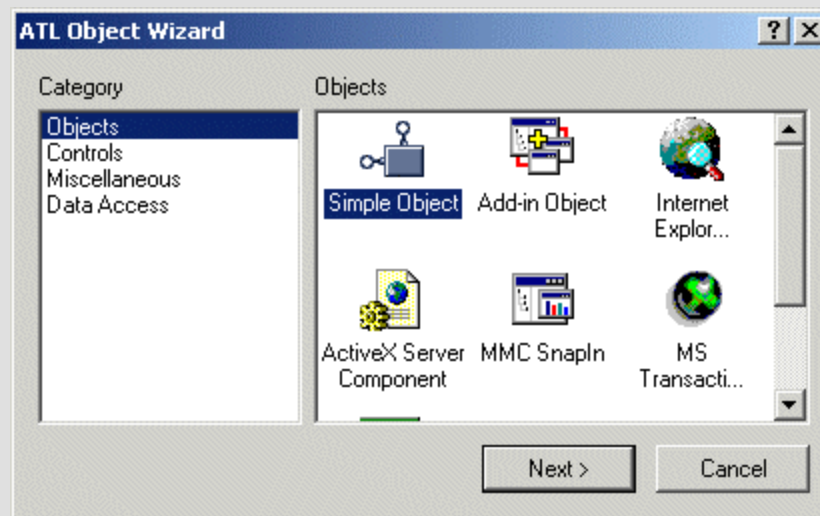
Passo 2

Seleccionar *New ATL Object...* . Este passo pode também ser feito através do menu principal, seleccionando “*New ATL Object*” na opção de menu *Insert*.

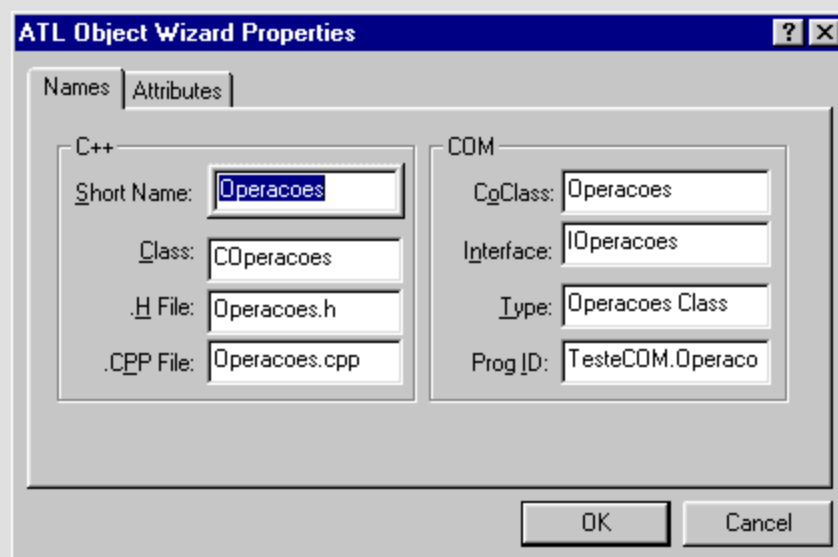


Passo 3

De seguida, aparece uma janela identificada por *ATL Object Wizard*. Selecciona-se, nesta janela, a categoria *Objects* e, escolhe-se como objecto *Simple Object*. Depois, carrega-se em *Next*.

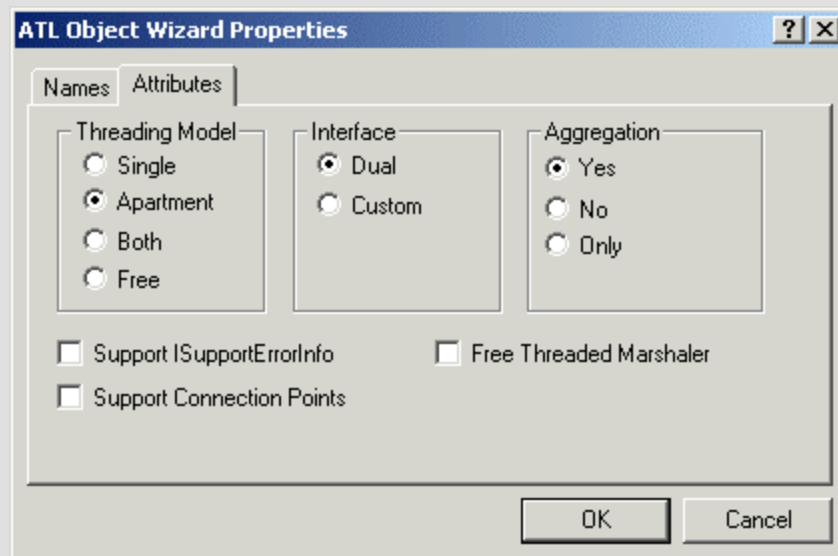
**Passo 4**

Na próxima janela, escolhemos a *Tab Names* e colocamos o nome do objecto na secção *Short Name*: – neste caso, colocar *Operacoes*. Todas as outras secções serão preenchidas automaticamente com os nomes por omissão.



Passo 5

Seleccionar a *Tab Attributes*, escolhendo como opções, *Apartment threading*, *Dual Interface* e *Agregation 'Yes'*. A opção *Agregation* não é importante para o servidor em causa



Nos atributos do objecto, podemos especificar o *modelo de thread*, o *tipo de interface* e a *agregação*. Do *modelo de thread* podemos destacar o seguinte:

O *Apartment* significa que o objecto pode ser utilizado por mais que uma *thread*, mas o subsistema COM garante que apenas a *thread* que criou o objecto chamará métodos desse objecto. No fundo, cada instância do objecto pertence a uma única *thread*. Deve ter-se em atenção, que através deste modelo é possível criar múltiplos objectos em *threads* diferentes, portanto, se forem utilizados dados globais nos métodos os acessos terão de ser feitos através de um mecanismo *thread-safe*. Aconselha-se, caso se pretenda utilizar dados globais a escolher a opção *single* para o *modelo de thread*.

Quando escolhemos o modelo *Apartment*, é necessário que a fabrica de classes (*class factory*) seja *multithreaded-safe*, mas não o próprio objecto. Como o ATL implementa o *class factory*, não nos temos de preocupar, porque este irá fazer tudo o que é necessário.

Para o *interface*, escolhemos a opção *Dual*. Esta escolha significa que o objecto pode ser chamado, quer por via do interface *custom* – *IUnknown* –, quer por via de *Automation* –

IDispatch -. As linguagens de *script* utilizam apenas interfaces *Automation*, portanto, ao seleccionarmos a opção *Dual*, permitimos ao nosso objecto ser utilizado pelas linguagens de *script*.

Na secção *Aggregation*, podemos dizer se o nosso objecto pode ser contido dentro de outros. O objecto torna-se mais flexível se poder ser agregado, portanto devemos seleccionar a opção *Yes*.

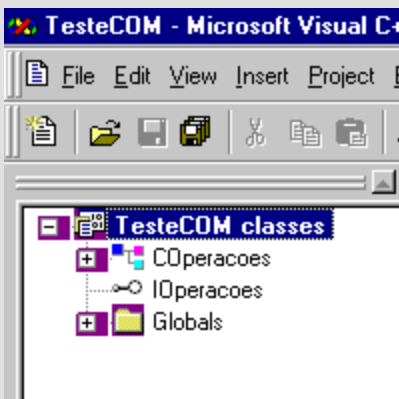
A opção *Support ISupportErrorInfo* serve para obter informações mais detalhadas de erros nos objectos COM, esta informação é muito utilizada pelo Visual Basic para descrever o erro com mais pormenor.

A opção *Support Connection Points*, serve para que o objecto possa disparar eventos.

A opção *Free Threaded Marshaler* é utilizada por certos tipos de *controls multithreaded*.

Passo 6

Carregar no botão "OK" para que o objecto COM seja criado



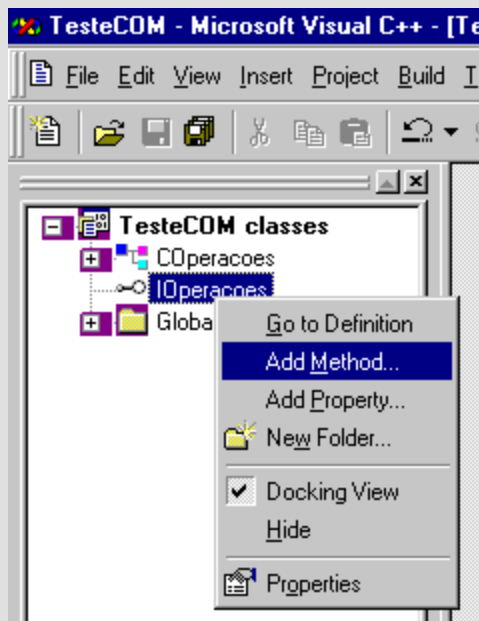
Adicionar um método ao Servidor recorrendo ao Wizard

Até ao momento, foi criado um objecto COM vazio, o que faz com que o Objecto seja inútil porque não consegue fazer nada. Irá, agora, ser criado um método simples, chamado *Adicao* que permitirá ao servidor receber dois valores inteiros longos e devolver a sua soma.

Passo 1

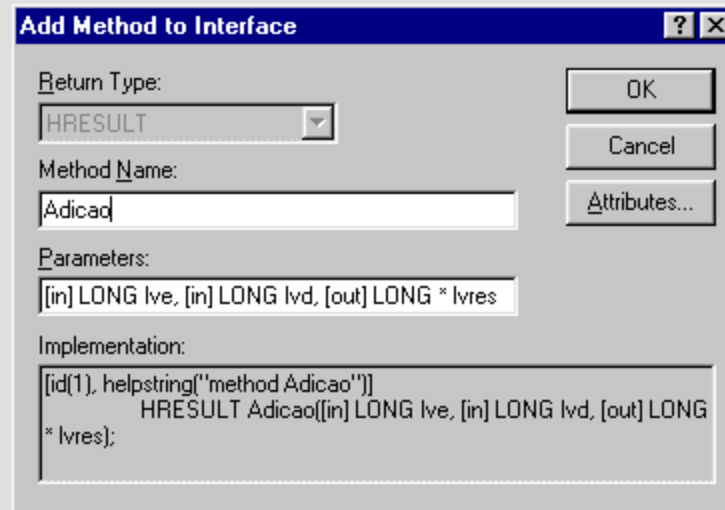
Na *tab Class View* selecciona-se o interface *IOperacoes*.

Carregar no botão do lado direito do rato, seleccionando a opção *Add Method* do menu.



Passo 2

Na janela *Add Method to Interface*, deve ser colocada a seguinte informação:



Add Method to Interface ? X

Return Type: HRESULT

Method Name: Adicao

Parameters: [in] LONG lve, [in] LONG lvd, [out] LONG * lvres

Implementation: [id(1), helpstring("method Adicao")] HRESULT Adicao([in] LONG lve, [in] LONG lvd, [out] LONG * lvres);

OK
Cancel
Attributes...

Após a adição do método e dos seus parâmetros, será criada a definição do MIDL. Esta definição é escrita em IDL e descreve o método ao compilador MIDL. Se pretendermos ver a definição do código associado ao método criado, basta, para tal, pressionar duas vezes o botão do rato sobre o interface *IOperacoes* na *Tab Class View*. Esta operação irá fazer com que o ficheiro *TesteCOM.idl* seja aberto. O que iremos ver será o seguinte:

```
...  
  
interface IOperacoes : IDispatch  
{  
    [id(1), helpstring("method Adicao")] HRESULT Adicao([in] LONG lve, [in] LONG lvd, [out] LONG * lvres);  
};  
...
```

O código apresentado é, de certa forma semelhante ao C++, sendo equivalente a um protótipo de uma função.

Se olharmos para os parâmetros do método *Adicao*, acabado de criar, podemos ver que cada tipo de dado é precedido de um atributo direccional: *in* ou *out*.

Estes atributos devem estar contidos entre parêntesis rectos e servem para especificar a direcção dos dados de cada parâmetro passado ao método, significando o seguinte:

In – parâmetro de *input*. Este pode também ser um apontador (por exemplo: **char ***), mas o valor por ele referenciado não pode ser retornado.

Out – parâmetro de *output*. O parâmetro tem de ser um apontador para uma zona de memória que irá receber o resultado.

retval – define um parâmetro que receberá o valor de retorno do método. Este atributo só pode ser utilizado associado ao último parâmetro do método. O parâmetro também terá de ter o atributo **out** e ser do tipo apontador. Exemplo: **[out, retval] BSTR * strOut**.

Passo 3

Devemos, agora, escrever o código em C++ para o método previamente definido. O *Wizard* já construiu uma estrutura vazia para o método, e acrescentou o protótipo do método à classe do objecto do servidor – *TesteCOM* - no ficheiro de *header* *TesteCOM.h*

O que devemos fazer, é abrir o *Operacoes.cpp* e procurar o método *Adicao*. De seguida, devemos introduzir o código representativo da funcionalidade que é requerida para o método. Neste caso, o que o método deverá fazer é somar os dois valores de entrada (*lve* e *lvd*) e colocar o resultado no argumento de saída *lvres*.

```
STDMETHODIMP COperacoes::Adicao(LONG lve, LONG lvd, LONG *lvres)
{
    *lvres = lve + lvd;

    return S_OK;
}
```

Finalmente, gravam-se as alterações e compila-se o projecto. A compilação faz com que seja criado o servidor COM: *TesteCOM.dll*.

Adicionar um método ao Servidor sem recorrer ao Wizard

O processo de adição de métodos ao objecto COM, pode ser feito recorrendo directamente à edição do ficheiro de idl – Interface definition language - e da classe que define o objecto. Supondo que pretendemos tornar o método Adicao mais genérico, ou seja, permitir que possam ser feitas adições de qualquer tipo de dado e não apenas de números inteiros longos. Se pretendemos adicionar o novo método recorrendo a este processo, os passos a seguir devem ser os seguintes:

Passo 1

No ficheiro TesteCOM.idl, no interface IOperacoes adicionar o novo método. O nome do método será AdicaoU

```
...
interface IOperacoes : IDispatch
{
[id(1), helpstring("method Adicao")] HRESULT Adicao([in] LONG lve, [in] LONG lvd, [out] LONG * lvres);

[id(2), helpstring("método de adição universal")] HRESULT AdicaoU([in] VARIANT lve, [in] VARIANT lvd, [out]
VARIANT * lvres);
};
...
```

Passo 2

Na classe COperacoes – ficheiro Operacoes.h – adicionar o protótipo para o novo método.

```
...
// IOperacoes
public:
STDMETHOD(Adicao)(/*[in]*/ LONG lve, /*[in]*/ LONG lvd, /*[out]*/ LONG * lvres);
STDMETHOD(AdicaoU)(/*[in]*/ VARIANT lve, /*[in]*/ VARIANT lvd, /*[out]*/ VARIANT * lvres);
};
...
```


Passo 3

No ficheiro Operacoes.cpp, definir o código para o método.

```

...
STDMETHODIMP Coperacoes::AdicaoU(VARIANT lve, VARIANT lvd, VARIANT *lvres)
{
    _variant_t _ve(lve);
    _variant_t _vd(lvd);

    VARTYPE t_ve=_ve.vt,
            t_vd=_vd.vt;
    _variant_t _vres;
    if( t_ve == VT_I2 && t_vd == VT_I2 ) // 2 byte signed int
    {
        _vres=_variant_t( long(_ve.iVal + _vd.iVal) );
        _vres.ChangeType( VT_I2 );
    }

    if( t_ve == VT_I4 && t_vd == VT_I4 ) // 4 byte signed int
    {
        _vres=_variant_t( long(_ve.iVal + _vd.iVal) );
        _vres.ChangeType( VT_I4 );
    }


    if( t_ve == VT_BSTR && t_vd == VT_BSTR ) // OLE Automation string
    {
        _vres=_variant_t( _bstr_t(lve) + _bstr_t(lvd) );
        _vres.ChangeType( VT_BSTR );
    }
    /* etc. */
    *lvres=_vres.Detach();
    return S_OK;
}

```

Finalmente, gravam-se as alterações e compila-se o projecto. A compilação faz com que seja criado o servidor COM, e que este passe a disponibilizar o método adicionado.

Ficheiros resultantes

Do projecto criado para a construção do servidor COM, resultam uma série de ficheiros. De seguida, será dada uma breve descrição sobre o que cada um representa:

TesteCOM.dsw	Define o ambiente do projecto.
TesteCOM.dsp	Informações sobre o projecto.
TesteCOM.plg	Ficheiro de log do projecto, que contém informação detalhada sobre a construção deste projecto.
TesteCOM.cpp	Rotinas principais da dll, implementações para que a dll possa ser exportada.
TesteCOM.h	Ficheiro gerado pelo MIDL que contém as definições para os interfaces.
TesteCOM.def	São declarados os métodos tradicionais da dll: <ul style="list-style-type: none"> • DllCanUnloadNow • DllGetClassObject • DllRegisterServer • DllUnregisterServer
TesteCOM.idl	O código idl do TesteCOM.dll, os ficheiros <i>idl</i> definem todos os componentes COM.
TesteCOM.rc	Ficheiro dos recursos. O principal recurso é o IDR_OPERACOES, que define o código utilizado para colocar a informação do COM no <i>Registry</i> . <div style="text-align: center;">  </div>
TesteCOM.tlb	Versão binária do ficheiro de idl, que disponibiliza a informação completa sobre os interfaces do objecto COM..
TesteCOM_i.c	Contém a informação dos <i>IID's - Interface Identifiers</i> - e dos <i>CLSID's - Classes Identifiers</i> -. Este ficheiro é normalmente incluído no ficheiro com o código (TesteCOM.cpp).
Operacoes.cpp	Implementação do COperacoes. Este ficheiro contém todo o código C++ para a definição dos métodos no objecto COM Exemplo.
Operacoes.h	Definição da classe do objecto COM Exemplo.
Operacoes.rgs	Código para o <i>Registry</i> . Utilizado para registar os componentes COM no mesmo.
StdAfx.cpp	O código para o <i>header</i> pré-compilado.
StdAfx.h	Ficheiro de <i>header</i> standard.

Com apenas alguns passos conseguimos criar um servidor COM completo. Se não fosse utilizado o *Wizard*, a escrita deste servidor demoraria horas para ser desenvolvida.

O código do servidor é praticamente todo gerado pelo *ATL Wizard*, e, por isso, este ambiente de trabalho é uma excelente forma de desenvolver rapidamente aplicações, devido a não termos de nos preocupar com os pormenores inerentes ao funcionamento do COM, mas sim com as questões relativas às funcionalidades que pretendemos disponibilizar pelo servidor.

Passos necessários à disponibilização do servidor:

1. Executar o compilador MIDL para gerar o código e as *type libraries*;
2. Compilar os ficheiros fonte;
3. Fazer o Link e criar o TesteCOM.dll;
4. Registrar a *dll* utilizando o utilitário **regsvr32** para que este servidor possa ser carregado sempre que solicitado.

Opções do regsvr32:

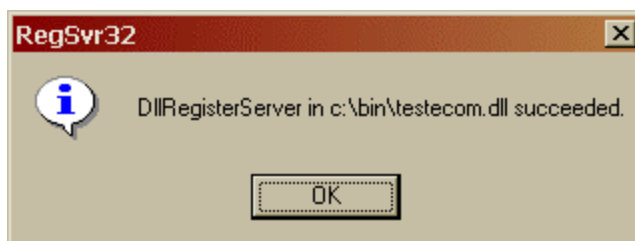
```
Usage: regsvr32 [/u] [/s] [/n] [/i[:cmdline]] dllname
/u -    Unregister server
/s -    Silent; display no message boxes
/i -    CallDllInstall passing it an optional [cmdline]; when used with /u calls dll uninstall
/n -    do not call DllRegisterServer; this option must be used with /i
```

Registo do servidor

Para registar o nosso servidor, e assumindo que o ficheiro do servidor se encontra do disco 'c:' no directório 'bin', devemos executar o seguinte comando:

```
C:\>regsvr32 "c:\bin\testecom.dll"
```

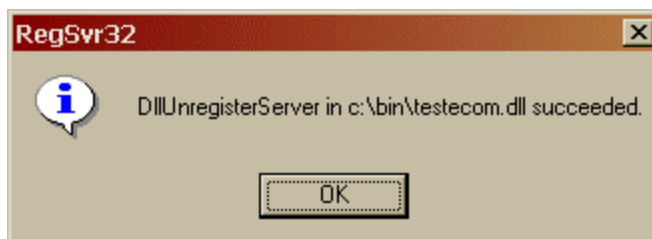
Se o servidor for bem registado, será exibida uma janela com informação do registo do mesmo:



Se pretendermos eliminar o registo do nosso servidor, temos que fazer uma operação inversa e o comando a ser executado deve ser o seguinte:

```
C:\>regsvr32 /u "c:\bin\testecom.dll"
```

Se a operação for bem sucedida, será mostrada uma janela com a seguinte informação:



O Cliente COM

Já vimos como é simples e rápido criar um servidor COM básico. A definição de um cliente COM é, de igual modo, extremamente simples de implementar. Com a utilização de um servidor e de um cliente COM conseguimos ter uma aplicação COM completa. Tendo em conta todas as características inerentes aos objectos COM, muitos programadores estão cada vez mais a começar a utilizar este meio para a criação de dll's. Tal deve-se à facilidade que é requerida para configurar um servidor *In-process* e começar a utilizá-lo de imediato.

De seguida, irá mostrar-se como se define uma aplicação, de forma a que esta se comporte como um cliente COM. A aplicação cliente será muito simples, tentando apenas mostrar a forma como se pode incluir suporte ao COM e como se processa a invocação de métodos que existem no servidor. Este exemplo irá utilizar o servidor COM, que foi criado anteriormente (TesteCOM.dll). Como é óbvio, quando estamos a falar de aplicações reais, estas contêm mais complexidade e mais funcionalidade.

Serão criadas duas aplicações cliente para o servidor COM desenvolvido. Uma será desenvolvida em Visual Basic e a outra em Visual C++.

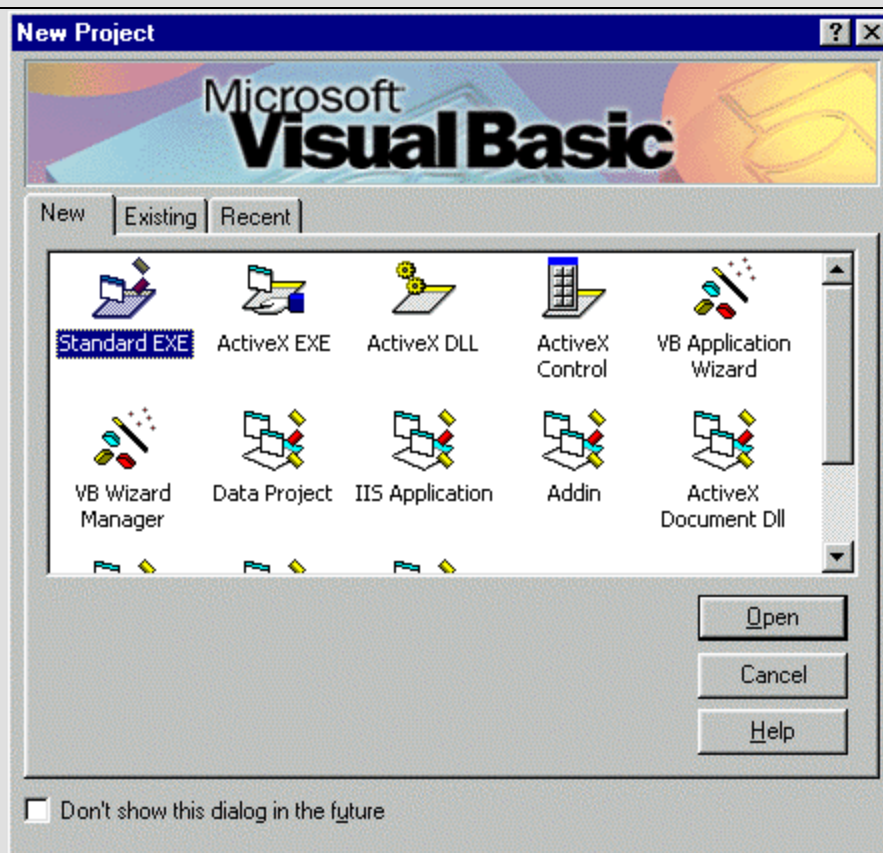
Cliente COM em Visual Basic

Esta aplicação terá o nome ClienteVB e utilizará o métodos *Adicao* e *AdicaoU* do servidor TesteCOM.

Os passos necessários à construção do cliente COM em Visual Basic são os seguintes:

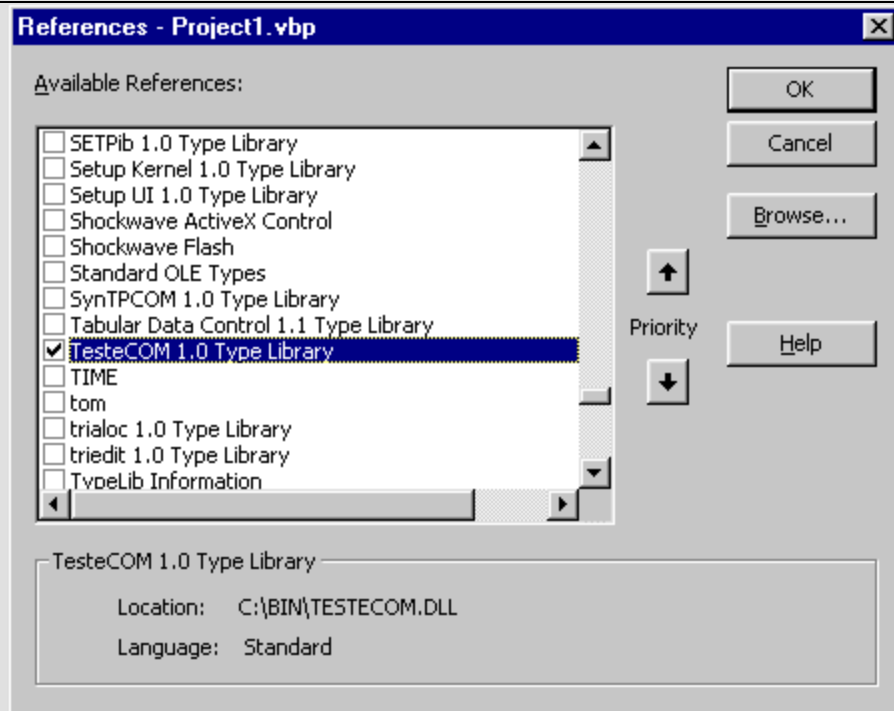
Passo 1

Criar um novo projecto do tipo Standard EXE.



Passo 2

Na opção Project|References... , seleccionar o servidor COM anteriormente criado.

**Passo 3**

No *Form* por omissão da aplicação, adicionar um ou mais botões para as opções que pretendemos criar. Para o exemplo em causa, devem ser criados três botões, um para invocar o método *Adicao*, e dois para invocarem o método *AdicaoU*.



Passo 4

Depois, deve-se digitar o código em Visual Basic para cada uma das funcionalidades que pretendemos. Neste caso, deve-se declarar e criar um objecto que aponte para o interface que pretendemos do servidor COM. Após termos o novo objecto, podemos invocar métodos do interface, ou seja, podemos invocar o método Adicao e AdicaoU.

```
Dim obj As New TESTECOMLib.Operacoes
```

```
Private Sub Adicao_Click()
```

```
    Dim resp As Long
```

```
    obj.Adicao 10, 20, resp
```

```
    MsgBox "Resultado: " + Str(resp)
```

```
End Sub
```

```
Private Sub AdicaoUnivInt_Click()
```

```
    Dim resp As Variant
```

```
    obj.AdicaoU 100, 200, resp
```

```
    MsgBox "Resultado: " + Str(resp)
```

```
End Sub
```

```
Private Sub AdicaoUnivTexto_Click()
```

```
    Dim resp As Variant
```

```
    obj.AdicaoU "abc", "def", resp
```

```
    MsgBox "Resultado: " + resp
```

```
End Sub
```

Como se pode verificar, a criação de um cliente COM em Visual Basic é extremamente simples, o que leva cada vez mais à utilização de objectos COM no desenvolvimento de aplicações. Também, será possível alterar e corrigir o servidor COM, sem que para isso haja a necessidade de alterações nos clientes.

Cliente COM em Visual C++

A Aplicação cliente a demonstrar terá o nome ClienteCOM e utilizará o servidor TesteCOM.dll previamente criado. Esta aplicação será criada a partir do Visual C++ e será do tipo *MFC AppWizard (exe)*. De seguida, será demonstrado passo-a-passo, a forma de criar a aplicação e incluir as funcionalidades desejadas para que possam ser invocados métodos no servidor. A interacção que a aplicação irá ter, será feita através de uma janela de diálogo, que permitirá, através da selecção de botões, invocar métodos no servidor COM.

Passo 1

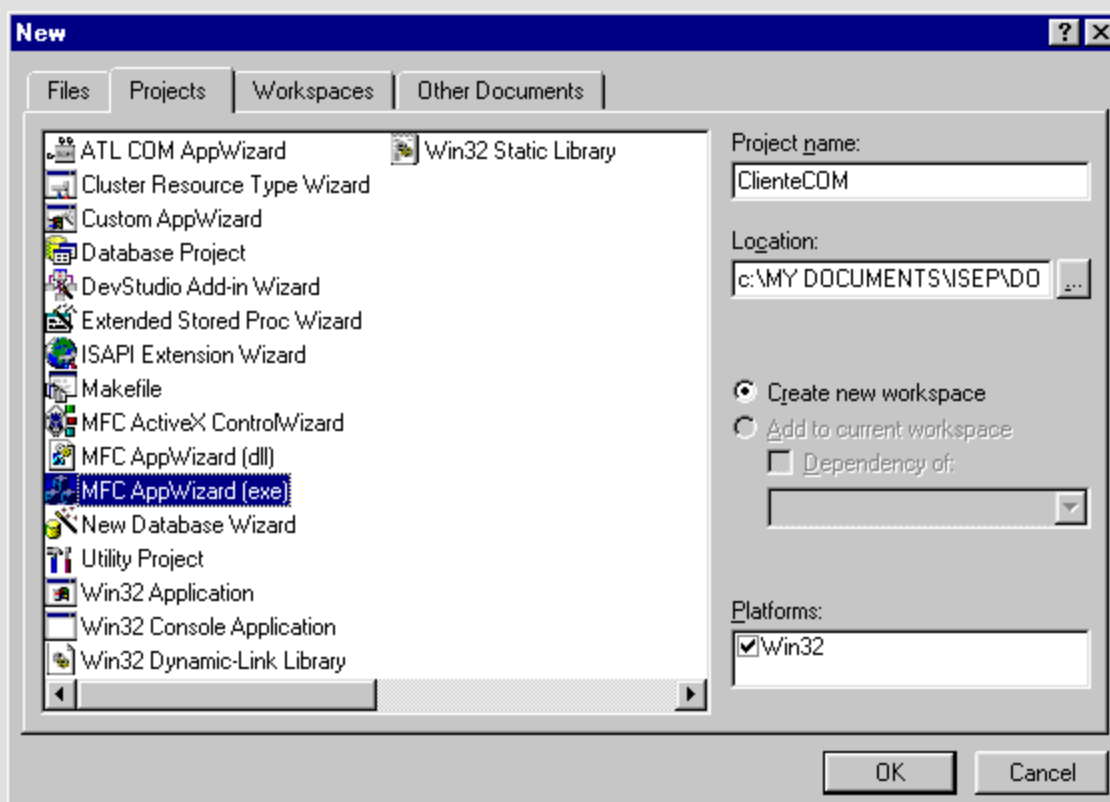
Seleccionar a opção File|New... do menu principal.



Passo 2

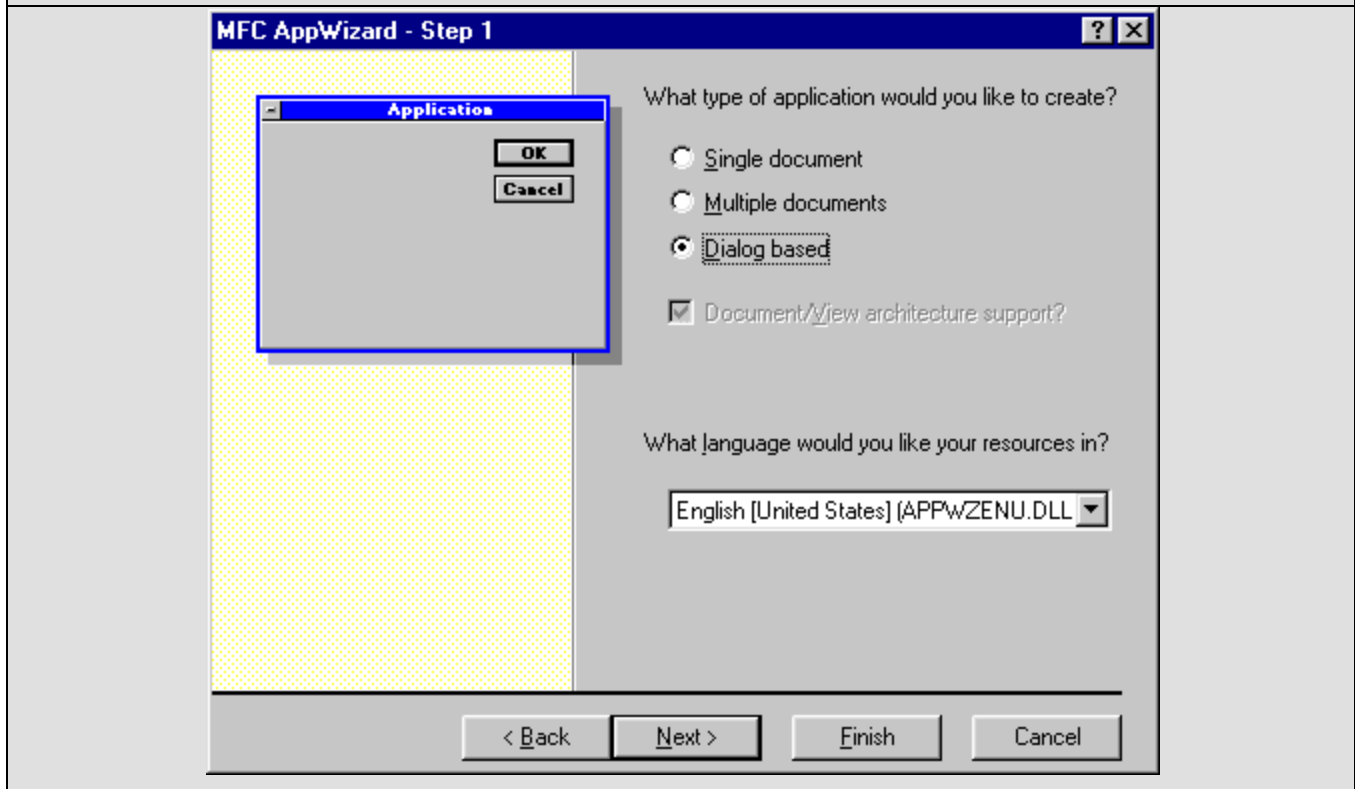
Seleccionar a *tab Projects* da janela *New*, e escolher *MFC AppWizard(exe)* da lista de projectos disponibilizada, e digitar o seguinte:

Project Name	ClienteCOM
Location	A que pretendemos para o nosso projecto
☉ Create New Workspace	



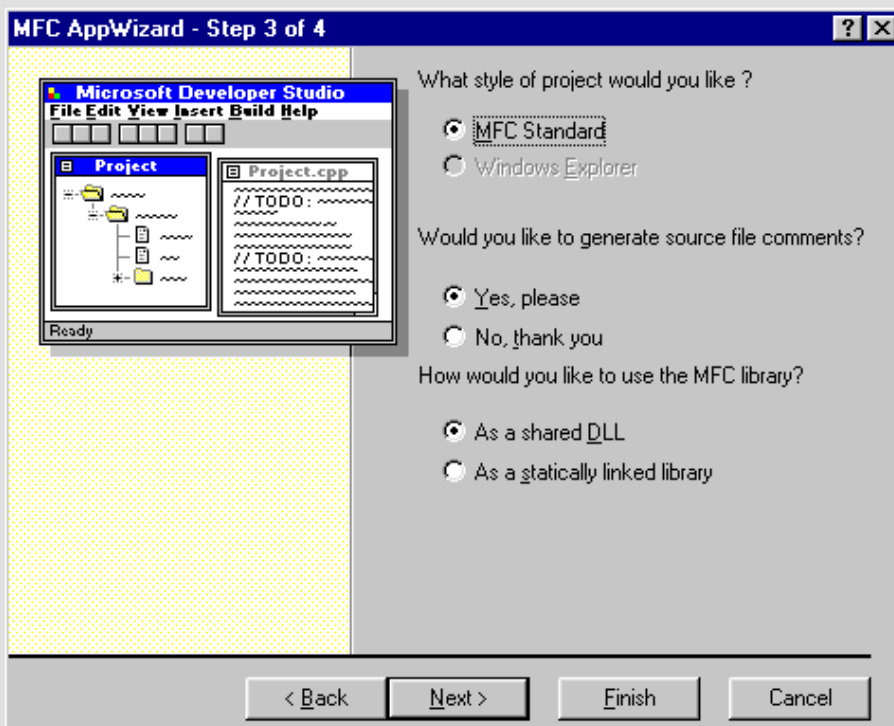
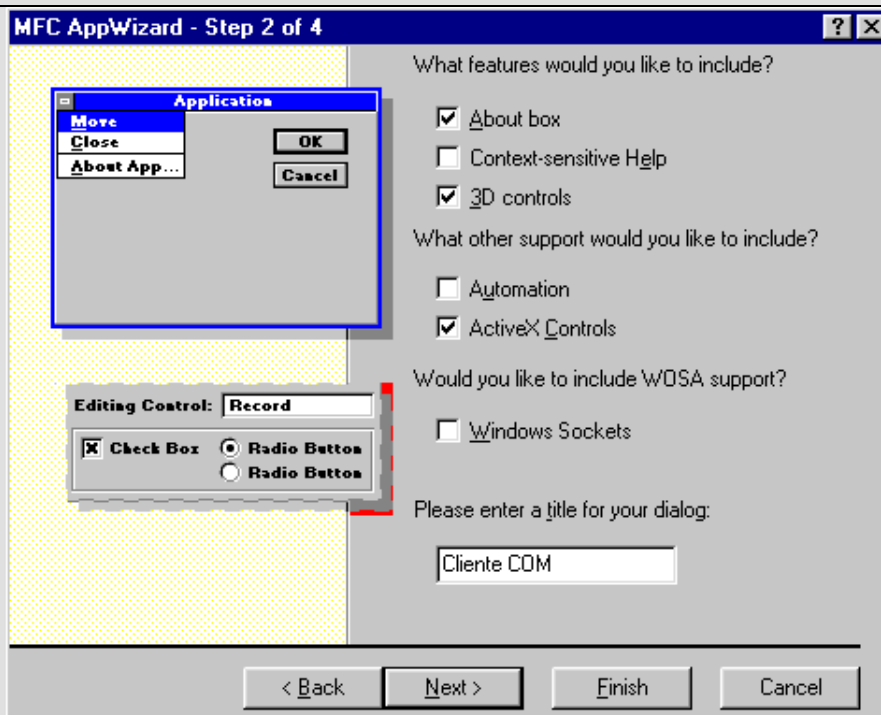
Passo 3

De seguida, há que seleccionar a opção que faz com que a aplicação seja *Dialog based*, isto é, uma aplicação cujo interface é baseado em janelas de diálogo.



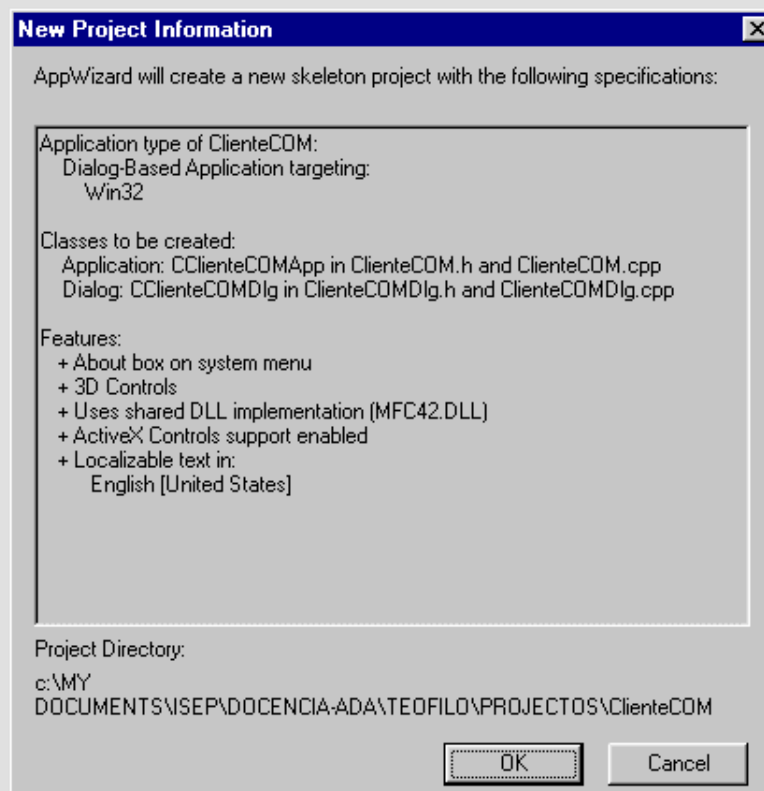
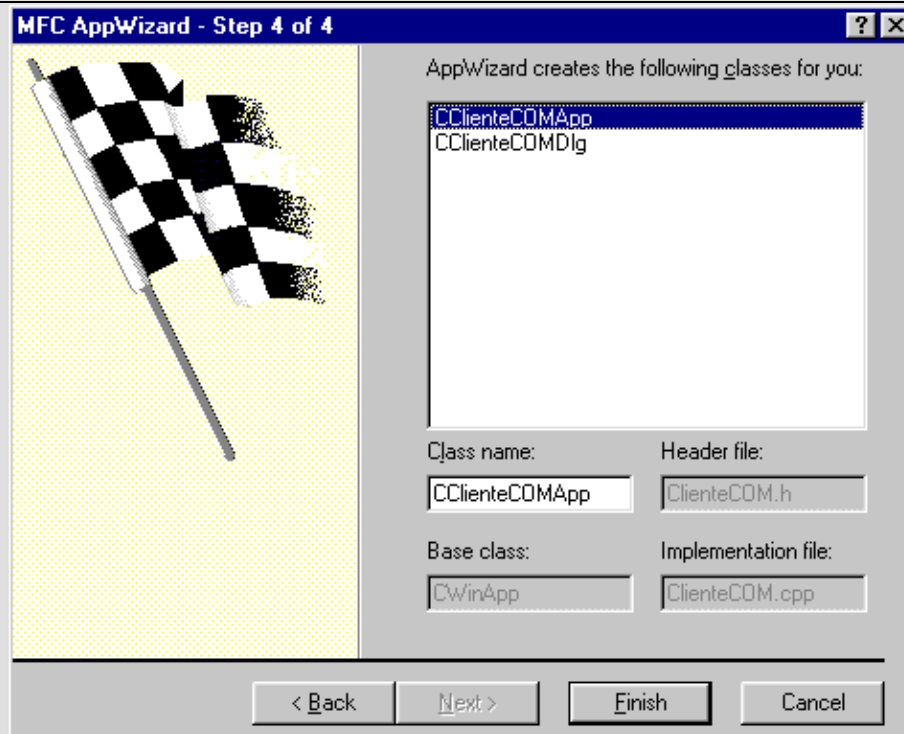
Passo 4

Nas janelas seguintes, podemos avançar no processo de definição da aplicação, assumindo os valores por omissão.



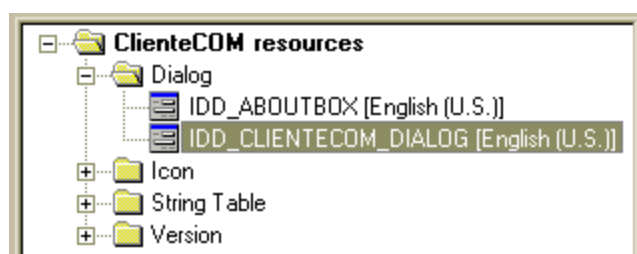
Passo 5

De seguida, podemos finalizar o processo, carregando no botão *Finish*. A esta janela segue-se uma outra final, em que é mostrado um resumo com a descrição da aplicação a ser criada.

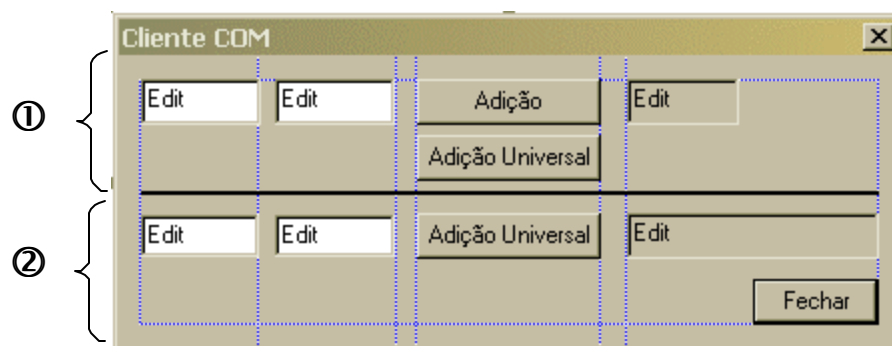


Após estes passos, o processo de definição da estrutura para a aplicação fica terminado. O *AppWizard* criará um projecto com todos os ficheiros necessários ao correcto funcionamento da aplicação. Embora esta aplicação compile sem erros e corra perfeitamente, ela apenas irá mostrar uma janela vazia sem qualquer tipo de funcionalidade. Para que a aplicação tenha utilidade, é necessário criar mecanismos de interacção com a mesma. Para o caso concreto, serão definidos na janela por defeito da aplicação, três botões para chamar os métodos do servidor COM.

O recurso associado à janela utilizada chama-se `IDD_CLIENTECOM_DIALOG` e está disponível na secção de recursos :



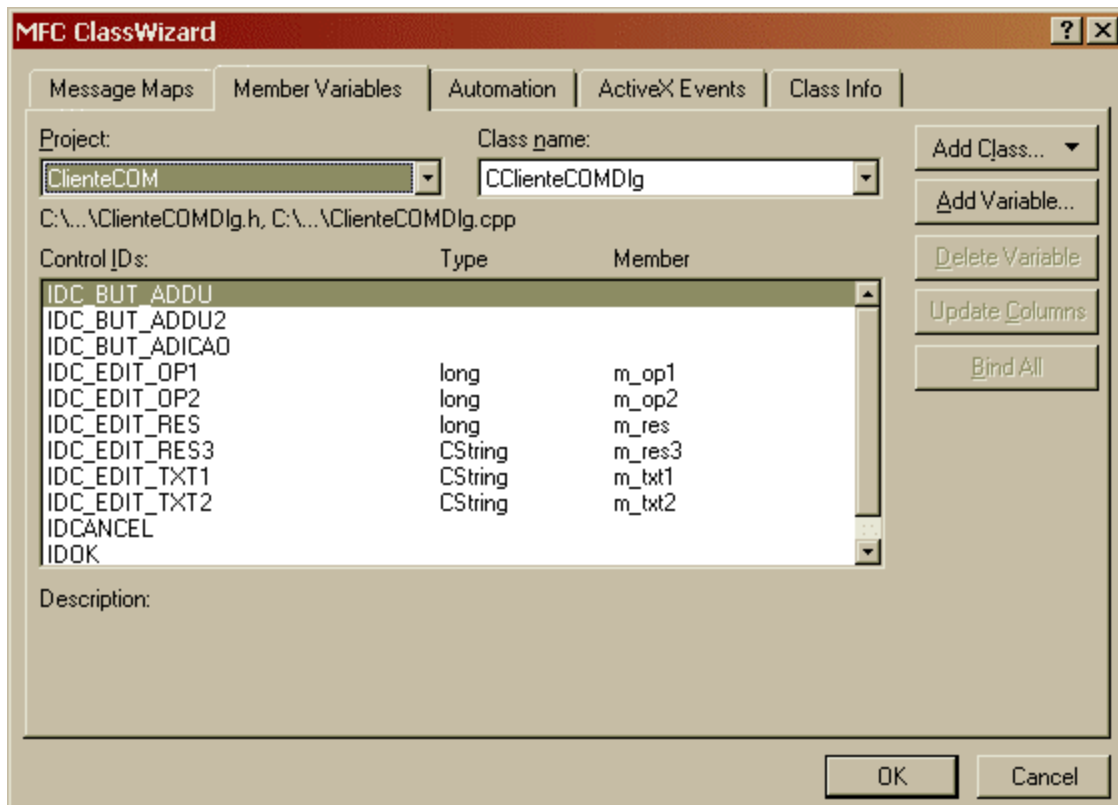
A janela associada ao recurso em questão terá o seguinte aspecto:



Neste caso, a secção ① mostra a utilização do método *Adicao* do servidor COM, método este, que toma os valores das caixas de edição como operandos e devolve o resultado da sua soma também para uma caixa de edição.

Podemos também ver a utilização do método *AdicaoU*, - nas secções ① e ② - que permite fazer somas, sejam os operandos valores numéricos ou alfanuméricos.

Para que se possa associar informação às caixas de edição da janela apresentada, deve-se associar variáveis para cada uma. Para que se faça esta associação, deve-se utilizar o *MFC ClassWizard*, e definir as variáveis de forma a produzir o seguinte:



Após esta associação, podemos utilizar as variáveis como argumentos que serão passados aos métodos *Adicao* e *AdicaoU* do servidor COM.

Fazer com que a aplicação seja um cliente COM

Passo 1

A aplicação cliente estabelece contacto com o servidor através da utilização de *Smart-Pointers* e, por conseguinte, recorre à directiva *import*. Para tal, o ficheiro `StdAfx.h` deve incluir a directiva *import*, com a definição do local onde se encontra o servidor

```
...  
  
// Dll do Servidor COM a utilizar na aplicação  
#import "TesteCOM.dll" //no_namespace  
  
...
```

Passo 2

Iniciar o subsistema COM, que pode ser feito pela definição de uma estrutura no ficheiro `ClienteCOM.cpp` da seguinte forma:

```
...  
  
CClienteCOMApp theApp;  
  
// iniciar o subsistema COM  
struct InitOle  
{  
    InitOle() {::CoInitialize(NULL);};  
    ~InitOle() {::CoUninitialize()};  
} _init_InitOle_  
  
...
```

A partir desta altura, a aplicação já está apta a criar objectos do servidor COM especificado, bem como a invocar métodos de interfaces do mesmo.

Definir métodos clientes que invoquem os métodos do servidor COM

Código associado ao botão Adição – secção ① :

```
void CClienteCOMDlg::OnButAdicao()
{
    UpdateData();
    ::TESTECOMLib::IOperacoesPtr spOper;

    try{
        TESTEHR( spOper.CreateInstance( __uuidof(::TESTECOMLib::Operacoes) ) );

        TESTEHR( spOper->Adicao( m_op1, m_op2, &m_res ) );

        UpdateData(FALSE);
    }
    catch(_com_error &e)
    {
        ::MessageBox(NULL,e.ErrorMessage(),"COM ERROR!",MB_ICONSTOP|MB_OK);
    }
}
```

Código associado ao botão Adição Universal – secção ① :

```
void CClienteCOMDlg::OnButAddu()
{
    UpdateData();
    ::TESTECOMLib::IOperacoesPtr spOper;
    VARIANT vRes;

    try{
        TESTEHR( spOper.CreateInstance( __uuidof(::TESTECOMLib::Operacoes) ) );

        TESTEHR( spOper->AdicaoU( m_op1, m_op2, &vRes ) );

        m_res= _variant_t( vRes );

        UpdateData(FALSE);
    }
    catch(_com_error &e)
    {
        ::MessageBox(NULL,e.ErrorMessage(),"COM ERROR!",MB_ICONSTOP|MB_OK);
    }
}
```

Código associado ao botão Adição Universal – secção ②:

```
void CClienteCOMDlg::OnButAddu2()
{
    UpdateData();
    ::TESTECOMLib::IOperacoesPtr spOper;
    VARIANT vRes;

    try{

        TESTEHR( spOper.CreateInstance( __uuidof(::TESTECOMLib::Operacoes) ) );
        TESTEHR(spOper->AdicaoU(m_txt1.AllocSysString(),m_txt2.AllocSysString(), &vRes ) );

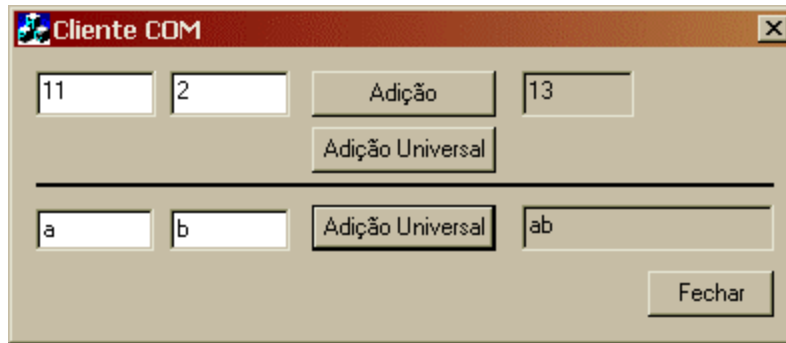
        m_res3= CString( _variant_t( vRes ).bstrVal );

        UpdateData(FALSE);

    }
    catch(_com_error &e)
    {
        ::MessageBox(NULL,e.ErrorMessage(),"COM ERROR!",MB_ICONSTOP|MB_OK);
    }
}
```

O aspecto mais interessante dos métodos, é a forma como é obtido o apontador para o interface *Operacoes*. Como se pode ver pelo código, a forma de obtenção do apontador é muito simples, pelo que basta declarar um apontador do tipo *IOperacoesPtr* e de seguida, através do *CreateInstance*, criar uma instancia desse interface. A partir desse momento, é possível invocar métodos que pertençam ao interface referido. Neste caso, é invocado o método *Adicao* e *AdicaoU*.

Um possível exemplo do nosso cliente:



Como se pode ver, a invocação de métodos de servidores COM, por aplicações clientes, é extremamente simples e poderosa. Uma vez definidos os servidores e os métodos necessários à funcionalidade requerida, podemos, através de um ou vários clientes, utilizar de forma transparente, essas funcionalidades. Ainda existe a vantagem de se poderem fazer alterações nos servidores –alteração ao código dos métodos, por exemplo, sem termos de compilar ou alterar o cliente.

Utilização de Componentes de Terceiros

Até aqui vimos como criar um servidor COM típico e como desenvolver aplicações cliente que acedem aos serviços disponibilizados por este.

Sendo a tecnologia COM, uma tecnologia cada vez mais utilizada para o desenvolvimento de aplicações, torna-se necessário mostrar, ainda que de uma forma muito superficial, como se podem aceder a objectos que são desenvolvidos por terceiros.

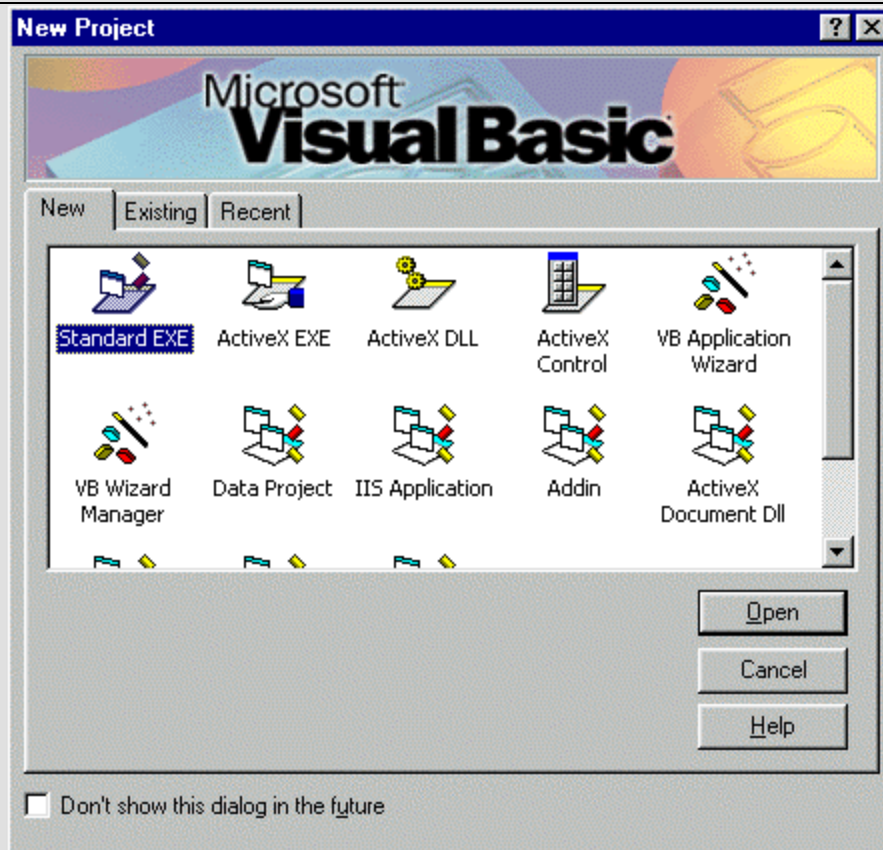
Para mostrar esta característica, vão ser desenvolvidas duas aplicações cliente idênticas, sendo uma em Visual Basic e a outra em Visual C++. As aplicações terão como objectivo criar um documento do Microsoft Word 2000.

Cliente em Visual Basic

A Aplicação cliente a demonstrar terá o nome wordVB e utilizará o servidor COM Microsoft Word 9.0 Object Library.

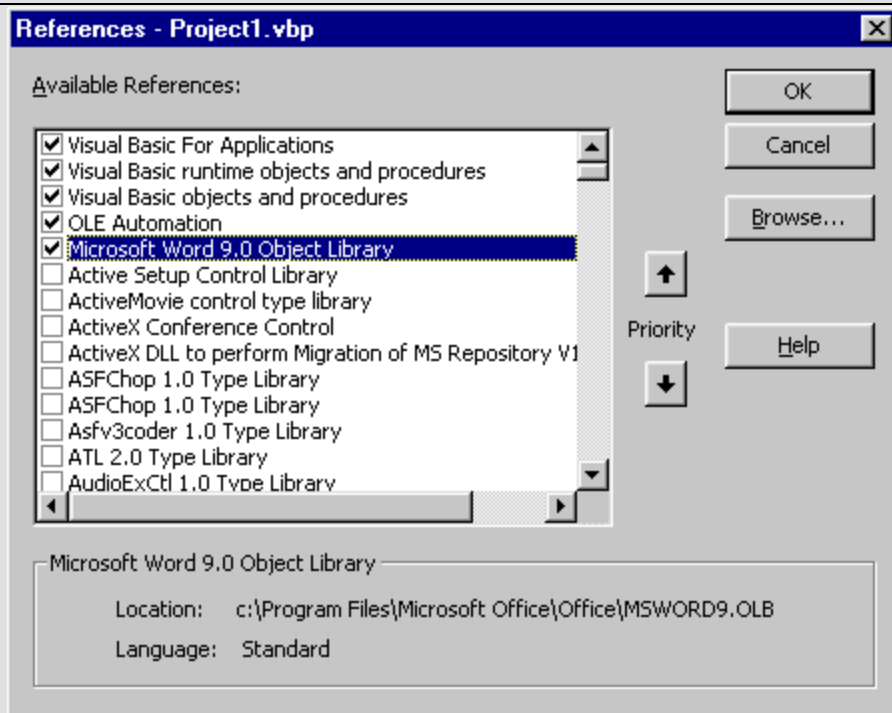
Passo 1

Criar um novo projecto do tipo Standard EXE.

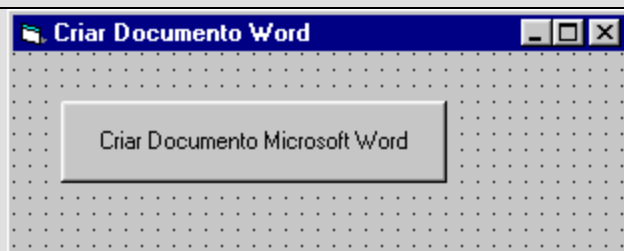


Passo 2

Na opção Project|References... , seleccionar o objecto do Microsoft Word 9.0.

**Passo 3**

No *Form* por defeito da aplicação, adicionar um botão para despoletar a funcionalidade requerida.



Passo 4

Depois, deve-se digitar o código que permite criar um documento exemplo no Microsoft Word 2000.

Dim oWordApp As Word.Application

```
Private Sub CriaDoc_Click()

'Criar um novo objecto do Microsoft Word
Set oWordApp = New Word.Application

With oWordApp

.Documents.Add 'Criar um novo documento

'Adicionar texto ao documento
.Selection.TypeText Text:="Documento de teste"
.Selection.TypeParagraph
.Selection.TypeText Text:="produzido pelo"
.Selection.TypeParagraph
.Selection.TypeText Text:="VB para a cadeira de ADAV."

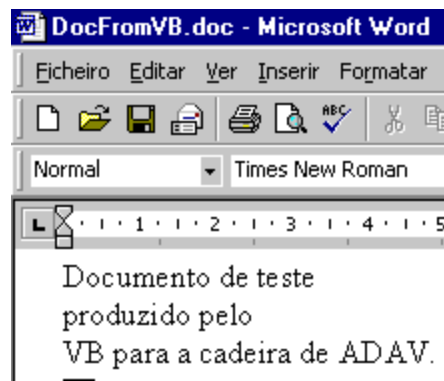
'Gravar o documento
.ActiveDocument.SaveAs FileName:=App.Path + "\DocFromVB.doc", _
FileFormat:=wdFormatDocument, LockComments:=False, _
Password:="", AddToRecentFiles:=True, WritePassword _
:="", ReadOnlyRecommended:=False, EmbedTrueTypeFonts:=False, _
SaveNativePictureFormat:=False, SaveFormsData:=False, _
SaveAsAOCELetter:=False

End With

oWordApp.Quit 'Sair do Word

End Sub
```

Após a execução do código acima descrito, será produzido um documento Microsoft Word com o seguinte conteúdo:

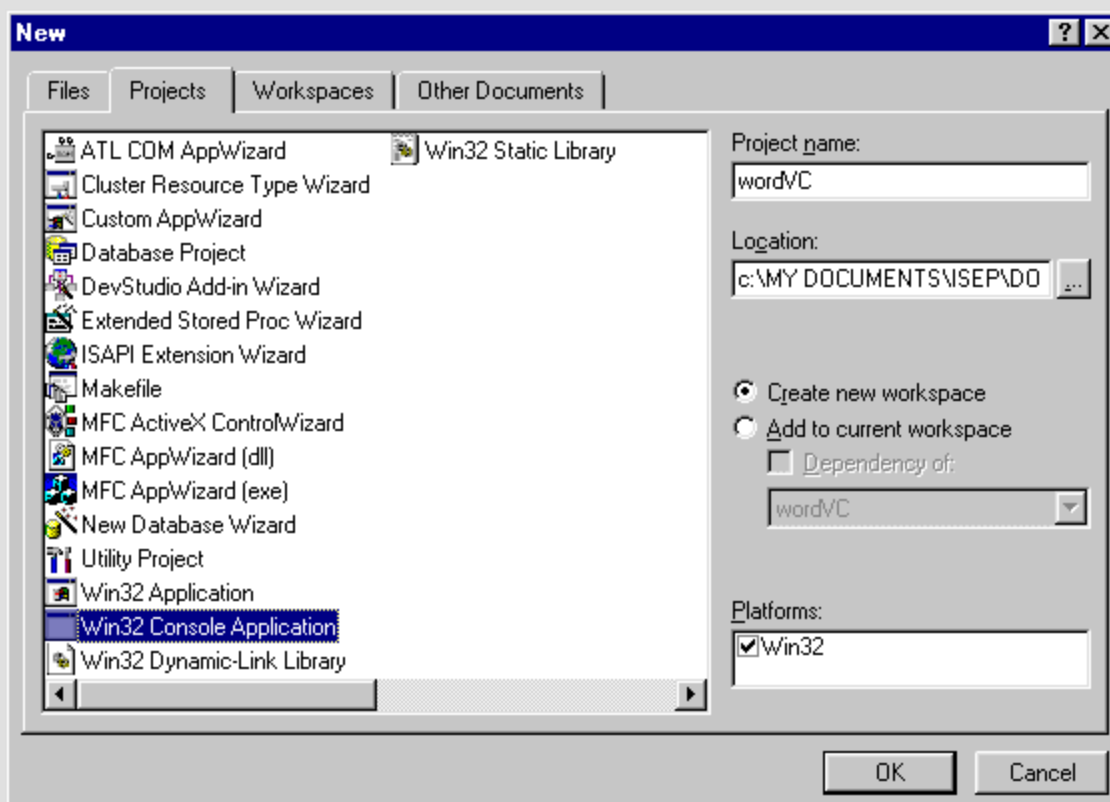


Cliente em Visual C++

A Aplicação cliente a demonstrar terá o nome wordVC e utilizará o servidor COM Microsoft Word 9.0 Object Library.

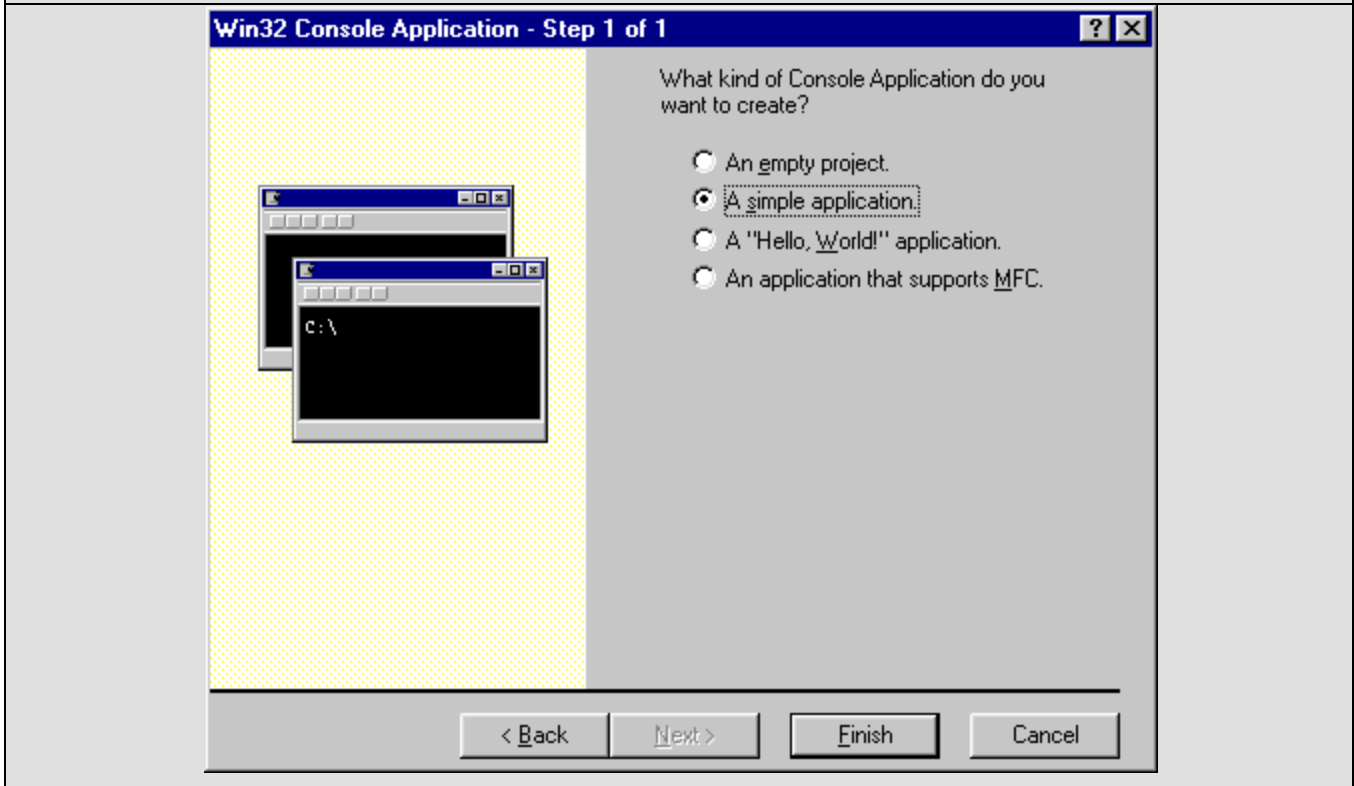
Passo 1

Seleccionar File|new... e na *tab Projects* da janela *New*, escolher Win32 Console Application e definir um nome para o projecto. Por exemplo:



Passo 2

De seguida, deve-se seleccionar a opção: A simple application:



Passo 3

A aplicação cliente estabelece contacto com os objectos do Microsoft Office através da utilização de *Smart Pointers* e, por conseguinte, recorre à directiva *import*. Para tal, o ficheiro StdAfx.h deve incluir a directiva *import*, com a definição do local onde se encontram os objectos.

```
...
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#import "C:\Program Files\Microsoft Office\Office\mso9.dll"
#import "C:\Program Files\Common Files\Microsoft Shared\vba\vba6\vbe6ext.olb"
#import "C:\Program Files\Microsoft Office\Office\msword9.olb" rename("ExitWindows", "_ExitWindows ")

#endif
```

Passo 4

Iniciar o subsistema COM, que pode ser feito pela definição de uma estrutura da seguinte forma:

```
...

// iniciar o subsistema COM
struct InitOle
{
    InitOle() {::CoInitialize(NULL);}
    ~InitOle() {::CoUninitialize();}
} _init_InitOle_;

...
```

Passo 5

Depois, deve-se digitar o código que permite criar um documento exemplo no Microsoft Word 2000.

```
#define TESTEHR(hr) if (FAILED(hr)) _com_issue_error(hr);

int main(int argc, char* argv[])
{
    ::Word::ApplicationPtr spWordApp;
    char theDir[1024];

    GetCurrentDirectory(1024,theDir);
    strcat(theDir,"\\DocFromVC.doc");

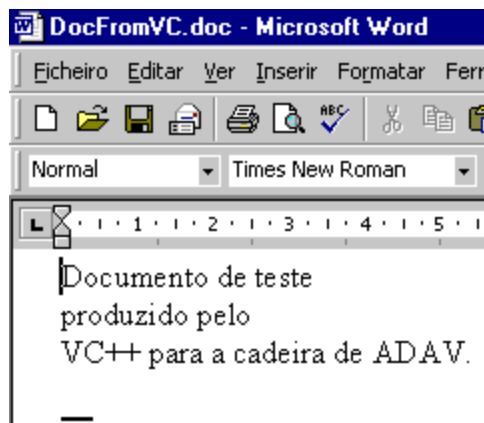
    try{
        TESTEHR( spWordApp.CreateInstance(__uuidof(::Word::Application)) );

        TESTEHR( spWordApp->Documents->Add() );
        TESTEHR( spWordApp->Selection->TypeText( "Documento de teste" ) );
        TESTEHR( spWordApp->Selection->TypeParagraph() );
        TESTEHR( spWordApp->Selection->TypeText( "produzido pelo" ) );
        TESTEHR( spWordApp->Selection->TypeParagraph() );
        TESTEHR( spWordApp->Selection->TypeText( "VC++ para a cadeira de ADAV." ) );
        TESTEHR( spWordApp->Selection->TypeParagraph() );
        TESTEHR( spWordApp->ActiveDocument->SaveAs( &_variant_t( theDir ).Detach() ) );

        TESTEHR( spWordApp->Quit() );
    }
    catch(_com_error &e)
    {
        ::MessageBox(NULL,_bstr_t(e.Error())+"\n"+e.ErrorMessage(),"COM", MB_ICONSTOP|MB_OK);
    }

    return 0;
}
```

Após a execução do código acima descrito, será produzido um documento Microsoft Word com o seguinte conteúdo:



Exercício de referência – Fase 3.

Chegado aqui, e assumindo que as noções básicas necessárias à construção de um servidor COM estão adquiridas, torna-se agora necessário desenvolver um servidor que permita disponibilizar funcionalidades à aplicação de referência da disciplina de ADAV. Este servidor terá, inicialmente, apenas um interface –Iproduto- e um método – TemStockSuficiente -, que permitirá saber se um determinado produto existe e, em caso afirmativo, se o stock requisitado pode ou não ser satisfeito.

Após a construção do servidor, terá de ser adicionado à nossa aplicação de referência o código necessário para que esta possa ser um cliente COM e possa criar objectos do nosso servidor. Estes passos já foram descritos na secção ‘Cliente COM em Visual C++’ em ‘Como fazer com que a aplicação seja um cliente COM’, passos 1 e 2.

Criar o servidor utilizando o Wizard-ATL

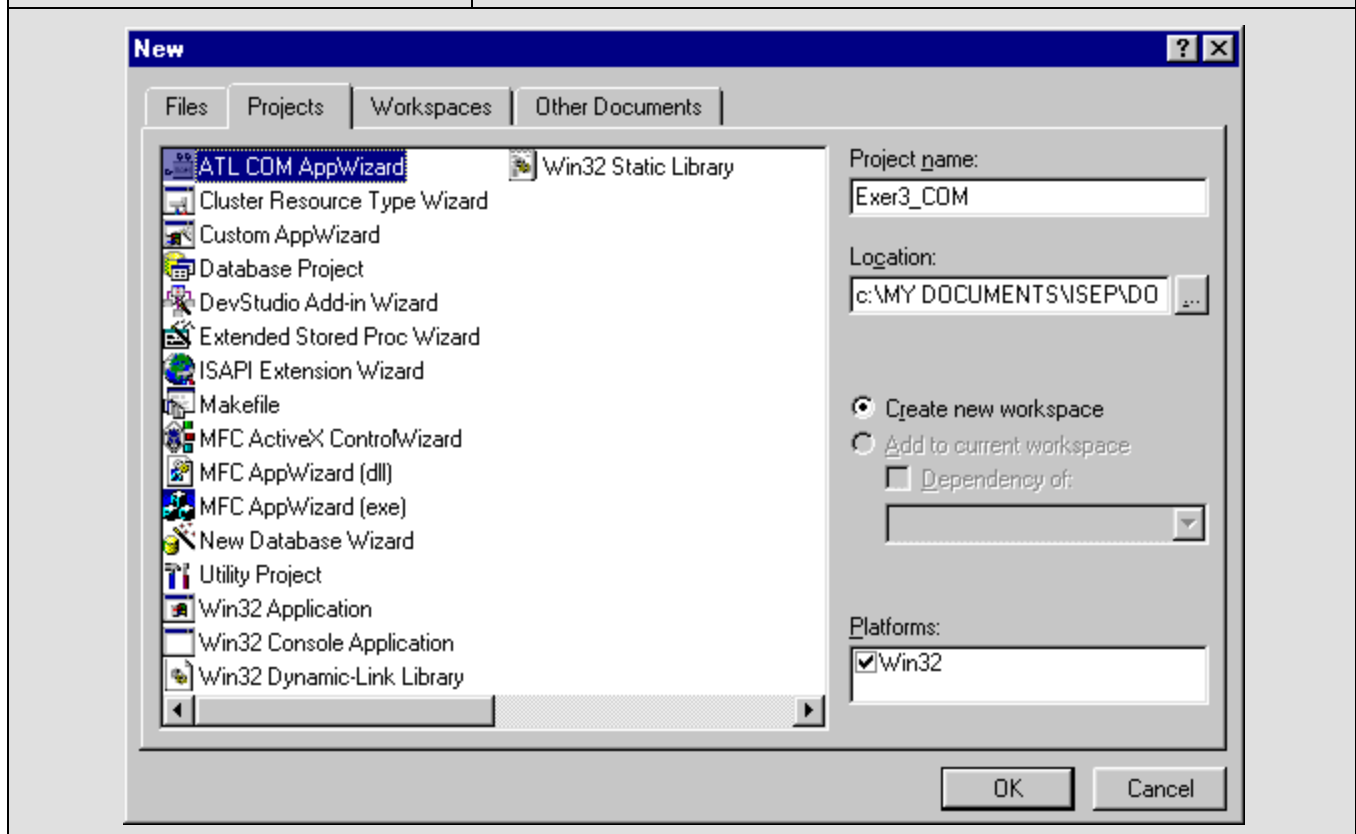
Passo 1

Primeiro seleccionamos a opção File|New... do menu principal.

Passo 2

De seguida seleccionamos a Tab *Projects* da janela *New*, e escolhemos **ATL COM AppWizard** da lista de projectos disponível e digitamos o seguinte:

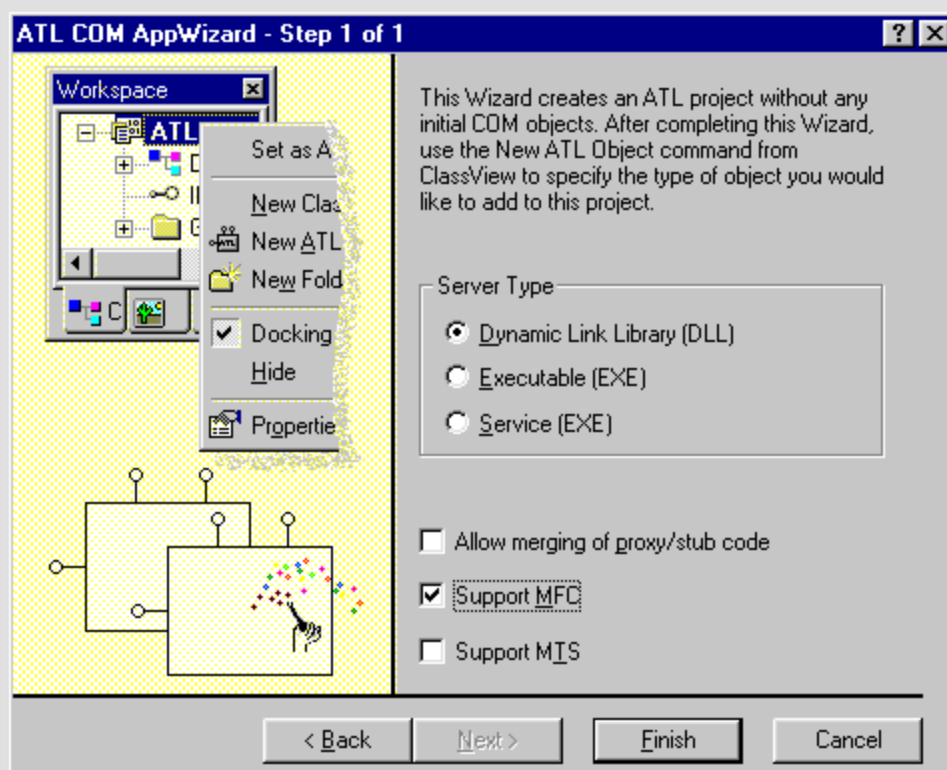
Project Name	Exer3_COM
Location	A que pretendemos para o nosso projecto
<input checked="" type="radio"/> Create New Workspace	



Passo 3

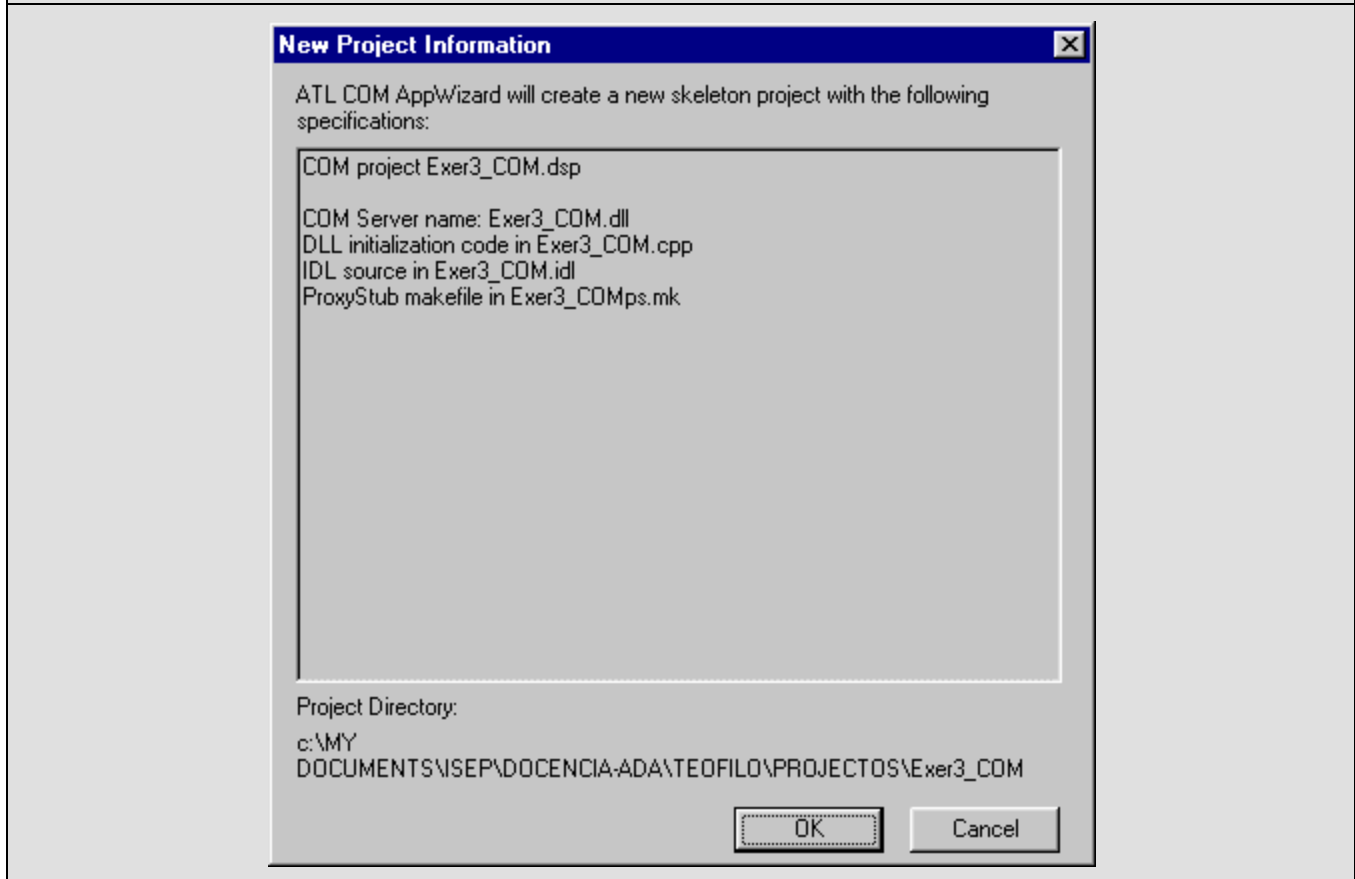
Na janela seguinte, devemos seleccionar a opção que permite criar um servidor *In-process* – dll -, e também especificar que é pretendido que o nosso servidor tenha suporte para as classes MFC.

Server Type Dynamic Link Library (DLL)

▼ Support MFC

Passo 4

Finalizar o processo, carregando no botão *Finish*.

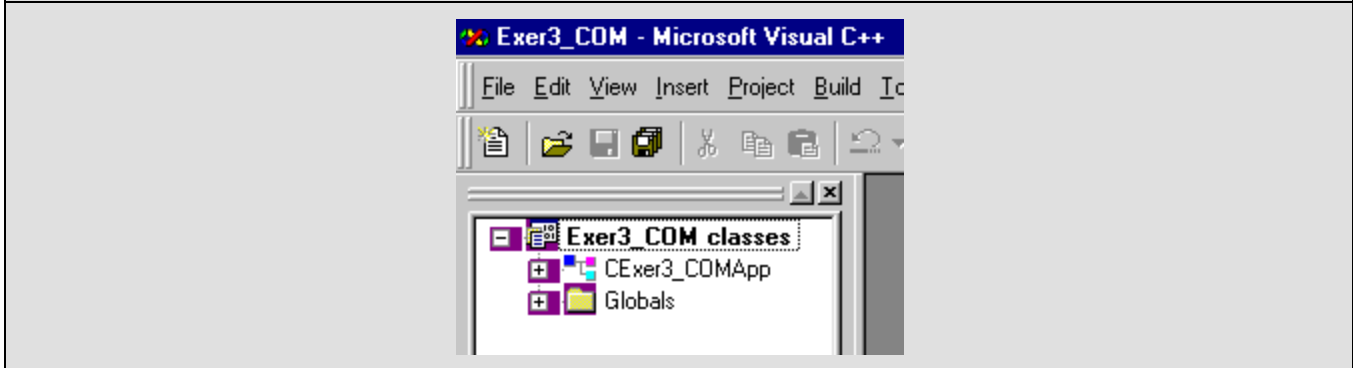


De seguida, vamos adicionar um objecto ao servidor, que disponibilizará um interface e uma classe que implementa esse interface.

Adicionar o objecto COM Produto

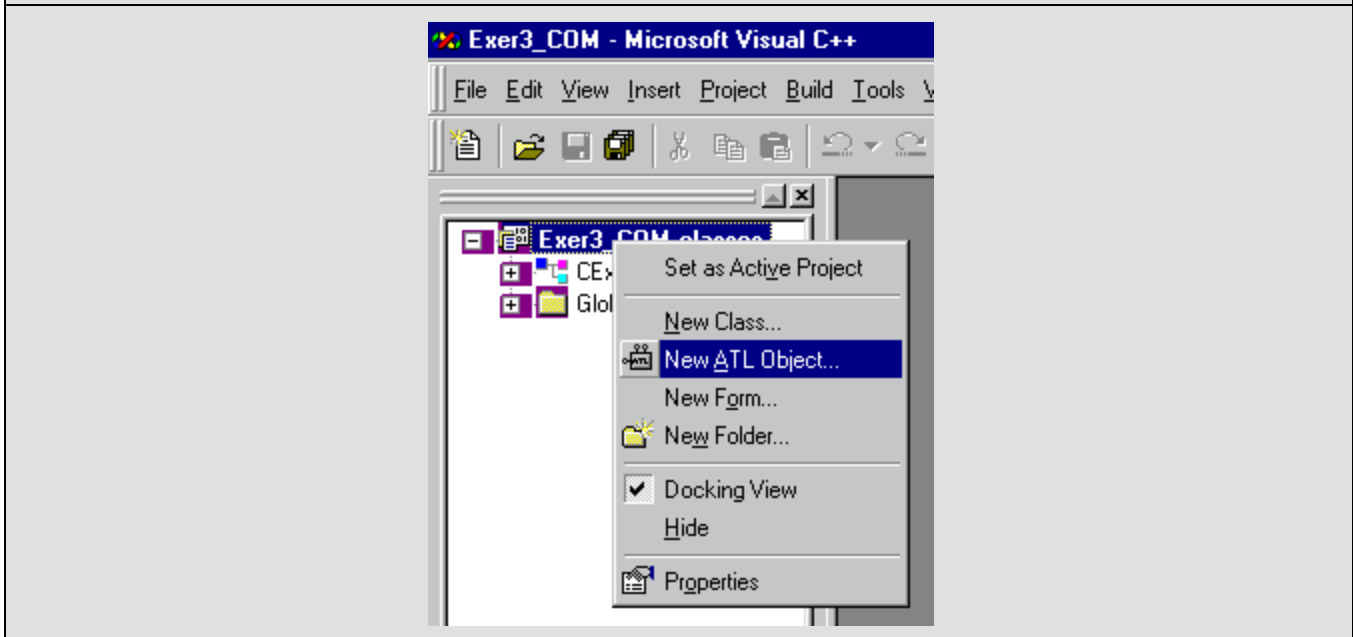
Passo 1

No *ClassView*, carregar no botão do lado direito do rato sobre o item Exer3_COM classes



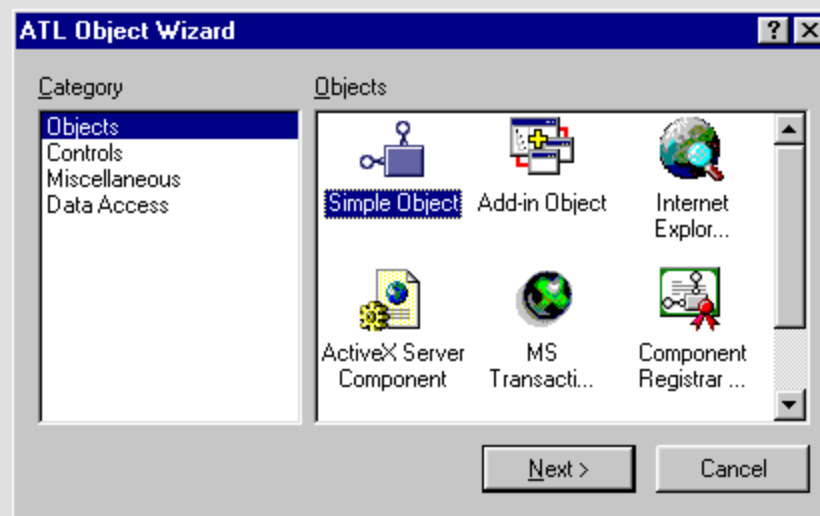
Passo 2

Seleccionar *New ATL Object...*. Este passo pode também ser feito através do menu principal, seleccionando *New ATL Object* na opção de menu *Insert*



Passo 3

De seguida, aparece uma janela identificada por *ATL Object Wizard*. Selecciona-se, nesta janela, a categoria *Objects* e escolhe-se, como objecto, *Simple Object*. Depois, carrega-se em *Next*.

**Passo 4**

Na próxima janela, escolhemos a *Tab Names* e colocamos o nome do objecto na secção *Short Name*: – neste caso colocar: *Produto*. Todas as outras secções serão preenchidas automaticamente com os nomes por omissão.

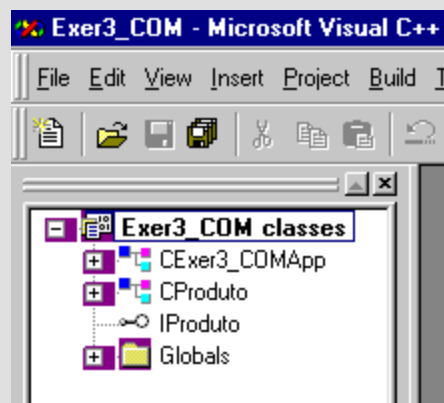


Passo 5

Selecione a *Tab Attributes*, escolhendo, como opções, *Apartment threading*, *Dual Interface* e *Agregation 'Yes'*. A opção *Agregation* não é importante para o servidor em causa.

**Passo 6**

Carregar no botão "OK" para que o objecto COM seja criado.



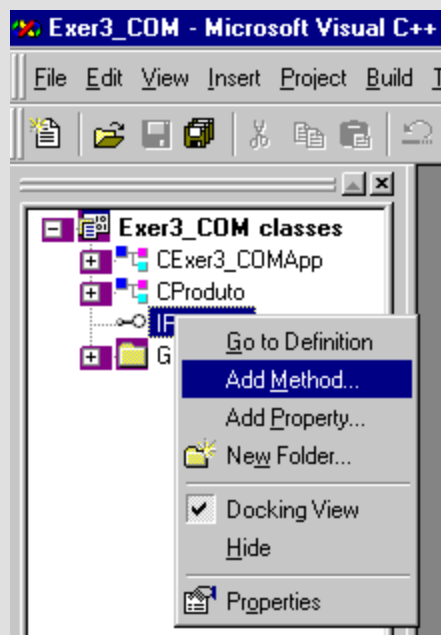
Adicionar o método *TemStockSuficiente* ao Servidor

Para que o novo objecto possa disponibilizar a funcionalidade pretendida, temos que adicionar um método ao interface.

Passo 1

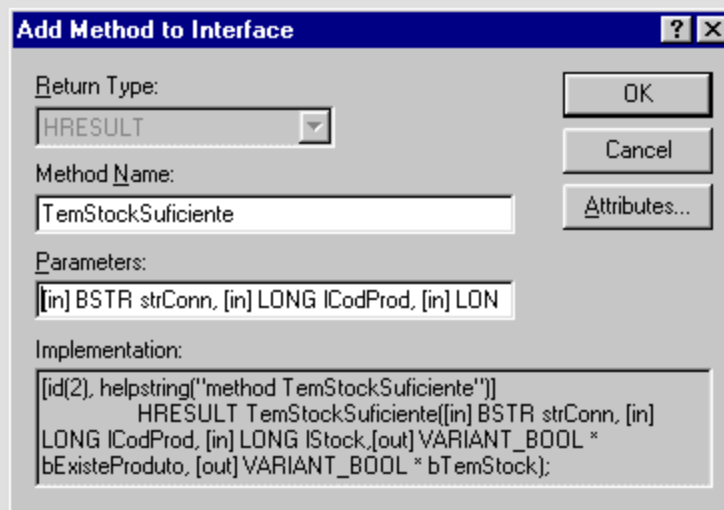
Na *tab Class View* selecciona-se o interface *IProduto*.

Carregar no botão do lado direito do rato, seleccionando a opção *Add Method* do menu.



Passo 2

Na janela *Add Method to Interface*, deve ser colocada a seguinte informação:



Add Method to Interface ? X

Return Type: HRESULT

Method Name: TemStockSuficiente

Parameters: [in] BSTR strConn, [in] LONG ICodProd, [in] LONG Istock

Implementation: [id(2), helpstring('method TemStockSuficiente')] HRESULT TemStockSuficiente([in] BSTR strConn, [in] LONG ICodProd, [in] LONG Istock, [out] VARIANT_BOOL * bExisteProduto, [out] VARIANT_BOOL * bTemStock);

OK
Cancel
Attributes...

Após a adição do método e dos seus parâmetros, será criada a definição do MIDL. Esta definição é escrita em IDL e descreve o método ao compilador MIDL. Para vermos o código IDL gerado pelo Wizard para a definição do servidor COM, basta abrir o ficheiro Exer3_COM.idl .

```
// Exer3_COM.idl : IDL source for Exer3_COM.dll
//
// This file will be processed by the MIDL tool to
// produce the type library (Exer3_COM.tlb) and marshalling code.

import "oidl.idl";
import "ocidl.idl";
[
    object,
    uuid(236B15E1-2286-4804-8360-57A22D2EBE4D),
    dual,
    helpstring("IProduto Interface"),
    pointer_default(unique)
]
interface IProduto : IDispatch
{
    [id(1), helpstring("method TemStockSuficiente")]
    HRESULT TemStockSuficiente([in] BSTR strConn, [in] LONG ICodProd,
    [in] LONG IStock,[out] VARIANT_BOOL * bExisteProduto,
    [out] VARIANT_BOOL * bTemStock);
};

[
    uuid(8EAE4B0C-99D4-4244-A0C0-29B2BC665290),
    version(1.0),
    helpstring("Exer3_COM 1.0 Type Library")
]
library EXER3_COMLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(36F2E8CB-A945-446E-BFA3-A960A3170199),
        helpstring("Produto Class")
    ]
    coclass Produto
    {
        [default] interface IProduto;
    };
};
```

Para vermos o código que fará o registo dos componentes COM no registry devemos abrir o ficheiro Produto.rgs .

```
HKCR
{
  Exer3_COM.Produto.1 = s 'Produto Class'
  {
    CLSID = s '{36F2E8CB-A945-446E-BFA3-A960A3170199}'
  }
  Exer3_COM.Produto = s 'Produto Class'
  {
    CLSID = s '{36F2E8CB-A945-446E-BFA3-A960A3170199}'
    CurVer = s 'Exer3_COM.Produto.1'
  }
  NoRemove CLSID
  {
    ForceRemove {36F2E8CB-A945-446E-BFA3-A960A3170199} = s 'Produto Class'
    {
      ProgID = s 'Exer3_COM.Produto.1'
      VersionIndependentProgID = s 'Exer3_COM.Produto'
      ForceRemove 'Programmable'
      InprocServer32 = s '%MODULE%'
      {
        val ThreadingModel = s 'Apartment'
      }
      'TypeLib' = s '{8EAE4B0C-99D4-4244-A0C0-29B2BC665290}'
    }
  }
}
```

As constantes com os GUID gerados na construção do nosso servidor podem ser encontradas no ficheiro Exer3_COM_i.c .

```
#ifdef __cplusplus
extern "C"{
#endif

#ifndef __IID_DEFINED__
#define __IID_DEFINED__

typedef struct _IID
{
    unsigned long x;
    unsigned short s1;
    unsigned short s2;
    unsigned char c[8];
} IID;

#endif // __IID_DEFINED__

#ifndef CLSID_DEFINED
#define CLSID_DEFINED
typedef IID CLSID;
#endif // CLSID_DEFINED

const IID IID_IProduto =
    {0x236B15E1,0x2286,0x4804,{0x83,0x60,0x57,0xA2,0x2D,0x2E,0xBE,0x4D}};

const IID LIBID_EXER3_COMLib =
    {0x8EAE4B0C,0x99D4,0x4244,{0xA0,0xC0,0x29,0xB2,0xBC,0x66,0x52,0x90}};

const CLSID CLSID_Produto =
    {0x36F2E8CB,0xA945,0x446E,{0xBF,0xA3,0xA9,0x60,0xA3,0x17,0x01,0x99}};

#ifdef __cplusplus
}
#endif
```

Este ficheiro resulta da compilação do *idl* pelo MIDL. Logo, para que este possa ser acedido, é necessário compilar o ficheiro Exer3_COM.idl .

Adicionar o código ao método *TemStockSuficiente*

Devemos, agora, escrever o código em C++ para o método previamente definido. O *Wizard* já construiu uma estrutura vazia para o método, e acrescentou o seu protótipo à classe do objecto do servidor no ficheiro de *header* Exer3_COM.h

O que temos que fazer, é abrir o Produto.cpp e procurar o método *TemStockSuficiente*. De seguida deve-se introduzir o seguinte código:

```
STDMETHODIMP CProduto::TemStockSuficiente(BSTR strConn, LONG lCodProd, LONG lStock,
VARIANT_BOOL * bExisteProduto, VARIANT_BOOL *bTemStock)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState())
    *bExisteProduto = VARIANT_TRUE;
    *bTemStock = VARIANT_TRUE;
    CDatabase db;
    CDBVariant varQt;
    char strQuery[256];
    CRecordset rs;

    try
    {
        if( ! db.OpenEx( (char*)_bstr_t(strConn) ) ) // com utilização de connection string
            return E_FAIL;
        rs.m_pDatabase = &db;

        sprintf(strQuery,"SELECT quantidadestock FROM PRODUTOS WHERE
            produtoid=%ld",lCodProd);

        if( ! rs.Open( CRecordset::snapshot , strQuery ) ) {
            db.Close();
            return E_FAIL;
        }

        // se não encontrou o registo retorna e avisa
        if( rs.GetRecordCount() == 0 )
            *bExisteProduto = VARIANT_FALSE;
        else{
            // obter o valor do campo
            rs.GetFieldValue("quantidadestock",varQt);

            // verificar se possui stock suficiente
            if( varQt.m_lVal < lStock )
                *bTemStock = VARIANT_FALSE;
        }
    }
    catch( CDBException* e ){ return E_FAIL; }

    // fechar os objectos
    if( rs.IsOpen() ) rs.Close();
    db.Close();}

```

No **stdAfx.h** devem-se incluir os seguintes ficheiros:

```
#include <afxdb.h>  
#include <comdef.h>
```

Finalmente, gravam-se as alterações e compila-se o projecto. A compilação faz com que seja criado o servidor **Exer3_COM.dll**.

Como invocar o método a partir da nossa aplicação cliente:

Para ilustrar a utilização do servidor COM criado, vão ser definidas duas funções que irão ser chamadas quando se estiver a adicionar uma nova venda e as caixas de edição relativas ao código do produto e da quantidade de produto perderem o *Focus*.

```
void C???View::OnKillfocusEditPro() // caixa de edição do código do produto perdeu o focus
{
    if (m_bEdit) { // se está em edição

        UpdateData(); // actualizar variáveis

        try
        {
            ::EXER3_COMLib::IProdutoPtr spProd;
            VARIANT_BOOL vtbExistePro , vtbDummy;

            TESTEHR( spProd.CreateInstance( __uuidof(::EXER3_COMLib::Produto) ) );

            TESTEHR(spProd->TemStockSuficiente( "DSN=lojaInform;UID=;PWD=",
            m_pSet->m_ProdutoID, 0, &vtbExistePro, &vtbDummy) );

            if( vtbExistePro == VARIANT_FALSE ) {

                ::MessageBox(NULL,"O Produto não existe!","Aviso.",MB_OK);

                // restabelecer o focus
                CEdit * pEdit = (CEdit*)GetDlgItem(IDC_EDIT_PRO);
                pEdit->SetFocus();
            }

            spProd=NULL;
        }
        catch(_com_error &e)
        {
            ::MessageBox(NULL,e.ErrorMessage(),"COM ERROR!",MB_ICONSTOP|MB_OK);
        }
    }
}
```

```
void C???View::OnKillfocusEditQtd()// caixa de edição da quantidade do produto perdeu o focus
{
    if (m_bEdit) { // se está em edição

        UpdateData(); // actualizar variáveis

        try
        {
            ::EXER3_COMLib::IProdutoPtr spProd;
            VARIANT_BOOL vtbExistePro , vtbTemStock;
            CEdit * pEdit;

            TESTEHR( spProd.CreateInstance( __uuidof(::EXER3_COMLib::Produto) ) );

            TESTEHR( spProd->TemStockSuficiente("DSN= lojaInform;UID=;PWD=",
m_pSet->m_ProdutoID, m_pSet->m_QuantidadeVendida, &vtbExistePro, &vtbTemStock) );

            if(vtbExistePro == VARIANT_FALSE){

                ::MessageBox(NULL,"O Produto não existe!","Aviso.",MB_OK);

                // restabelecer o focus
                pEdit = (CEdit*)GetDlgItem(IDC_EDIT_PRO);
                pEdit->SetFocus();
            }
            else{

                if(vtbTemStock == VARIANT_FALSE){

                    ::MessageBox(NULL,"Não existe stock suficiente!","Aviso.",MB_OK);

                    // restabelecer o focus
                    pEdit = (CEdit*)GetDlgItem(IDC_EDIT_QTD);
                    pEdit->SetFocus();
                }
            }

            spProd=NULL;
        }
        catch(_com_error &e)
        {
            ::MessageBox(NULL,e.ErrorMessage(),"COM ERROR!",MB_ICONSTOP|MB_OK);
        }
    }
}
```

Conclusão

Com este documento, tentou fazer-se uma breve introdução ao COM, mostrando os aspectos mais “importantes” no desenvolvimento de servidores e clientes COM.

Espera-se que, após a leitura deste documento e a consequente construção dos exemplos nele contidos, que o aluno esteja apto a desenvolver os seus próprios servidores COM e, posteriormente, criar aplicações cliente que acedam a esses servidores.

Também foi mostrado como se pode utilizar componentes de terceiros (por exemplo objectos do Microsoft Office), o que permite dar mais flexibilidade aos nossos clientes. Por exemplo poderíamos aceder ao objecto do Microsoft Outlook, para enviar um e-mail a um fornecedor, no sentido de requisitar mais quantidade de um determinado produto, assim que o seu stock atingisse um determinado valor mínimo.

Este documento resulta de uma compilação de várias fontes, de entre as quais destaco a MSDN da Microsoft.

ANEXOS

OLE - Object Linking and Embedding

O COM disponibiliza um conjunto de funções que estão na base das comunicações utilizadas pelo OLE. Estas funções encontram-se agrupadas em interfaces.

O que é o OLE

É um mecanismo que permite a escrita de programas que deixam outras aplicações fazerem edição de dados que elas não sabem tratar, por exemplo um editor de texto pode permitir a edição gráfica de imagens através de outra aplicação.

Este mecanismo também permite que sejam desenvolvidas aplicações que sirvam para editar dados dentro de outras aplicações.

Um objecto pertencente a um programa pode aparecer num outro programa de duas formas:

- Um objecto de um documento externo pode ser ligado (***linked***) ao documento de outro programa, em que neste caso, o objecto externo não é gravado como fazendo parte do documento do programa, mas apenas uma referência é gravada para permitir que o objecto seja obtido da sua origem.
- Um documento externo pode também ser inserido (***embedded***) no documento actual, em que neste caso ele é gravado conjuntamente com o documento actual.

A um documento que contenha um outro objecto inserido (***embedded***) ou referenciado (***linked OLE***) é chamado de '***compound document***'.

A vantagem de um ***linked object*** prende-se com o facto de este poder ser alterado independentemente do ***compound document***, portanto, quando se abre um documento que contém um ***linked object***, a última versão do objecto será automaticamente incorporada.

Contudo, se o ficheiro que contém o **linked object** for eliminado ou mesmo se o ficheiro for mudado de directório, o **compound document** não saberá do **linked object** e não será capaz de o localizar.

Com um **emdedded object**, este apenas existe no contexto de um **compound document** e, por isso, não é acessível independentemente. O **compound document** e os seus **embedded OLE objects** são um único ficheiro, não sendo possível perderem-se os objectos. Do ponto de vista do utilizador, não existe qualquer diferença de aspecto na utilização do **compound document**.

ActiveX

É uma tecnologia, assente no COM, que permite a componentes interagirem uns com os outros independentemente da linguagem em que foram escritos.

ActiveX control

É um componente reutilizável ActiveX que expõe as suas propriedades e métodos a clientes ActiveX.

Para que se possam utilizar estes controls, estes devem ser carregados em control containers, tais como o Microsoft Visual Basic, Microsoft Internet Explorer, etc. Normalmente são conhecidos como OLE control ou simplesmente como –OCX.

Automation

É uma tecnologia baseada no COM que permite aos programadores desenvolverem aplicações que exponham as suas funcionalidades a linguagens de script ou linguagens interpretadas. Com a utilização do Automation, torna-se possível estabelecer comunicação em *run time* com métodos e propriedades do objecto.

Cliente Automation

Trata-se de uma aplicação, ferramenta de programação, ou linguagem de script que pode aceder a serviços disponibilizados por objectos Automation.

MIDL - Microsoft Interface Definition Language

Define os interfaces entre os programas cliente e servidores. O compilador MIDL, permite aos programadores criarem ficheiros com a definição dos interfaces e configurar aplicações que possam fazer chamadas a funções remotas (RPC) e a interfaces do COM/DCOM. O MIDL também suporta a geração de *Type Libraries* para o *OLE Automation*.

BSTR

A BSTR, known as basic string or binary string, is a pointer to a wide character string used by Automation data manipulation functions.

```
typedef OLECHAR FAR* BSTR;
```

_bstr_t

Microsoft Specific

A _bstr_t object encapsulates the **BSTR** data type. The class manages resource allocation and deallocation, via function calls to **SysAllocString** and **SysFreeString**, and other **BSTR** APIs when appropriate. The _bstr_t class uses reference counting to avoid excessive overhead.

```
#include <comdef.h>
```

[Compiler COM Support Class Overview](#)

Construction

_bstr_t	Constructs a <u>_bstr_t</u> object.
-------------------------	-------------------------------------

Operations

copy	Constructs a copy of the encapsulated BSTR .
length	Returns the length of the encapsulated BSTR .

Operators

operator =	Assigns a new value to an existing <u>_bstr_t</u> object.
operator +=	Appends characters to the end of the <u>_bstr_t</u> object.
operator +	Concatenates two strings.
operator !	Checks if the encapsulated BSTR is a NULL string.
operator ==, !=, <, >, <=, >=	Compares two <u>_bstr_t</u> objects.
operator wchar_t*, char*	Extract the pointers to the encapsulated Unicode or multibyte BSTR object.

VARIANT and VARIANTARG

Use VARIANTARG to describe arguments passed within DISPPARAMS, and VARIANT to specify variant data that cannot be passed by reference. When a variant refers to another variant by using the VT_VARIANT | VT_BYREF vartype, the variant being referred to cannot also be of type VT_VARIANT | VT_BYREF. VARIANTS can be passed by value, even if VARIANTARGS cannot. The following definition of VARIANT is described in OAIIDL.H automation header file:

```
typedef struct FARSTRUCT tagVARIANT VARIANT;
typedef struct FARSTRUCT tagVARIANT VARIANTARG;

typedef struct tagVARIANT {
    VARTYPE vt;
    unsigned short wReserved1;
    unsigned short wReserved2;
    unsigned short wReserved3;
    union {
        Byte                bVal;                // VT_UI1.
        Short               iVal;                // VT_I2.
        long                 lVal;                // VT_I4.
        float               fltVal;              // VT_R4.
        double               dblVal;              // VT_R8.
        VARIANT_BOOL        boolVal;             // VT_BOOL.
        SCODE                scode;              // VT_ERROR.
        CY                   cyVal;              // VT_CY.
        DATE                 date;               // VT_DATE.
        BSTR                 bstrVal;            // VT_BSTR.
        DECIMAL              FAR* pdecVal;       // VT_BYREF|VT_DECIMAL.
        IUnknown             FAR* punkVal;       // VT_UNKNOWN.
        IDispatch            FAR* pdispVal;      // VT_DISPATCH.
        SAFEARRAY            FAR* parray;        // VT_ARRAY *.
        Byte                 FAR* pbVal;         // VT_BYREF|VT_UI1.
        short                FAR* piVal;         // VT_BYREF|VT_I2.
        long                 FAR* plVal;         // VT_BYREF|VT_I4.
        float                FAR* pfltVal;       // VT_BYREF|VT_R4.
        double               FAR* pdblVal;       // VT_BYREF|VT_R8.
        VARIANT_BOOL        FAR* pboolVal;      // VT_BYREF|VT_BOOL.
        SCODE                FAR* pscode;        // VT_BYREF|VT_ERROR.
        CY                   FAR* pcyVal;        // VT_BYREF|VT_CY.
        DATE                 FAR* pdate;         // VT_BYREF|VT_DATE.
        BSTR                 FAR* pbstrVal;      // VT_BYREF|VT_BSTR.
        IUnknown             FAR* FAR* ppunkVal; // VT_BYREF|VT_UNKNOWN.
        IDispatch            FAR* FAR* ppdispVal; // VT_BYREF|VT_DISPATCH.
        SAFEARRAY            FAR* FAR* pparray;  // VT_ARRAY *.
        VARIANT              FAR* pvarVal;       // VT_BYREF|VT_VARIANT.
        void                 FAR* byref;         // Generic ByRef.
        char                 cVal;               // VT_I1.
        unsigned short       uiVal;              // VT_UI2.
        unsigned long        ulVal;              // VT_UI4.
        int                  intval;             // VT_INT.
        unsigned int         uintVal;            // VT_UINT.
        char FAR *           pcVal;              // VT_BYREF|VT_I1.
        unsigned short FAR * puiVal;            // VT_BYREF|VT_UI2.
        unsigned long FAR * pulVal;             // VT_BYREF|VT_UI4.
        int FAR *           pintVal;            // VT_BYREF|VT_INT.
        unsigned int FAR *  puintVal;           //VT_BYREF|VT_UINT.
    };
};
```

To simplify extracting values from VARIANTARGs, Automation provides a set of functions for manipulating this type. Use of these functions is strongly recommended to ensure that applications apply consistent coercion rules.

The *vt* value governs the interpretation of the union as follows:

Value	Description
VT_EMPTY	No value was specified. If an optional argument to an Automation method is left blank, do not pass a VARIANT of type VT_EMPTY. Instead, pass a VARIANT of type VT_ERROR with a value of DISP_E_PARAMNOTFOUND.
VT_EMPTY VT_BYREF	Not valid.
VT_UI1	An unsigned 1-byte character is stored in <i>bVal</i> .
VT_UI1 VT_BYREF	A reference to an unsigned 1-byte character was passed. A pointer to the value is in <i>pbVal</i> .
VT_UI2	An unsigned 2-byte integer value is stored in <i>uiVal</i> .
VT_UI2 VT_BYREF	A reference to an unsigned 2-byte integer was passed. A pointer to the value is in <i>puiVal</i> .
VT_UI4	An unsigned 4-byte integer value is stored in <i>ulVal</i> .
VT_UI4 VT_BYREF	A reference to an unsigned 4-byte integer was passed. A pointer to the value is in <i>pulVal</i> .
VT_UINT	An unsigned integer value is stored in <i>uintVal</i> .
VT_UINT VT_BYREF	A reference to an unsigned integer value was passed. A pointer to the value is in <i>puintVal</i> .
VT_INT	An integer value is stored in <i>intVal</i> .
VT_INT VT_BYREF	A reference to an integer value was passed. A pointer to the value is in <i>pintVal</i> .
VT_I1	A 1-byte character value is stored in <i>cVal</i> .
VT_I1 VT_BYREF	A reference to a 1-byte character was passed. A pointer to the value is in <i>pcVal</i> .
VT_I2	A 2-byte integer value is stored in <i>iVal</i> .
VT_I2 VT_BYREF	A reference to a 2-byte integer was passed. A pointer to the value is in <i>piVal</i> .
VT_I4	A 4-byte integer value is stored in <i>lVal</i> .
VT_I4 VT_BYREF	A reference to a 4-byte integer was passed. A pointer to the value is in <i>plVal</i> .
VT_R4	An IEEE 4-byte real value is stored in <i>fltVal</i> .
VT_R4 VT_BYREF	A reference to an IEEE 4-byte real value was passed. A pointer to the value is in <i>pfltVal</i> .
VT_R8	An 8-byte IEEE real value is stored in <i>dblVal</i> .
VT_R8 VT_BYREF	A reference to an 8-byte IEEE real value was passed. A pointer to its value is in <i>pdblVal</i> .

VT_CY	A currency value was specified. A currency number is stored as 64-bit (8-byte), two's complement integer, scaled by 10,000 to give a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. The value is in <i>cyVal</i> .
VT_CY VT_BYREF	A reference to a currency value was passed. A pointer to the value is in <i>pcyVal</i> .
VT_BSTR	A string was passed; it is stored in <i>bstrVal</i> . This pointer must be obtained and freed by the BSTR functions, which are described in Conversion and Manipulation Functions .
VT_BSTR VT_BYREF	A reference to a string was passed. A BSTR* that points to a BSTR is in <i>pbstrVal</i> . The referenced pointer must be obtained or freed by the BSTR functions.
VT_DECIMAL	Decimal variables are stored as 96-bit (12-byte) unsigned integers scaled by a variable power of 10. VT_DECIMAL uses the entire 16 bytes of the Variant.
VT_DECIMAL VT_BYREF	A reference to a decimal value was passed. A pointer to the value is in <i>pdecVal</i> .
VT_NULL	A propagating null value was specified. (This should not be confused with the null pointer.) The null value is used for tri-state logic, as with SQL.
VT_NULL VT_BYREF	Not valid.
VT_ERROR	An SCODE was specified. The type of the error is specified in <i>scodee</i> . Generally, operations on error values should raise an exception or propagate the error to the return value, as appropriate.
VT_ERROR VT_BYREF	A reference to an SCODE was passed. A pointer to the value is in <i>pscode</i> .
VT_BOOL	A 16 bit Boolean (True/False) value was specified. A value of 0xFFFF (all bits 1) indicates True; a value of 0 (all bits 0) indicates False. No other values are valid.
VT_BOOL VT_BYREF	A reference to a Boolean value. A pointer to the Boolean value is in <i>pbool</i> .
VT_DATE	A value denoting a date and time was specified. Dates are represented as double-precision numbers, where midnight, January 1, 1900 is 2.0, January 2, 1900 is 3.0, and so on. The value is passed in <i>date</i> . This is the same numbering system used by most spreadsheet programs, although some specify incorrectly that February 29, 1900 existed, and thus set January 1, 1900 to 1.0. The date can be converted to and from an MS-DOS representation using VariantTimeToDosDateTime , which is discussed in Conversion and Manipulation Functions .
VT_DATE VT_BYREF	A reference to a date was passed. A pointer to the value is in <i>pdate</i> .
VT_DISPATCH	A pointer to an object was specified. The pointer is in <i>pdispVal</i> . This object is known only to implement IDispatch . The object can be queried as to whether it supports any other desired interface by calling

	QueryInterface on the object. Objects that do not implement IDispatch should be passed using VT_UNKNOWN.
VT_DISPATCH VT_BYREF	A pointer to a pointer to an object was specified. The pointer to the object is stored in the location referred to by <i>ppdispVal</i> .
VT_VARIANT	Invalid. VARIANTARGs must be passed by reference.
VT_VARIANT VT_BYREF	A pointer to another VARIANTARG is passed in <i>pvarVal</i> . This referenced VARIANTARG, <i>pvarVal</i> , cannot be another VT_VARIANT VT_BYREF. This value can be used to support languages that allow functions to change the types of variables passed by reference.
VT_UNKNOWN	A pointer to an object that implements the IUnknown interface is passed in <i>punkVal</i> .
VT_UNKNOWN VT_BYREF	A pointer to the IUnknown interface is passed in <i>ppunkVal</i> . The pointer to the interface is stored in the location referred to by <i>ppunkVal</i> .
VT_ARRAY <anything>	An array of data type <anything> was passed. (VT_EMPTY and VT_NULL are invalid types to combine with VT_ARRAY.) The pointer in <i>pparray</i> points to an array descriptor, which describes the dimensions, size, and in-memory location of the array. The array descriptor is never accessed directly, but instead is read and modified using the functions described in Conversion and Manipulation Functions .

Built on Monday, August 21, 2000

_variant_t

Microsoft Specific

A **_variant_t** object encapsulates the [VARIANT](#) data type. The class manages resource allocation and deallocation, and makes function calls to **VariantInit** and **VariantClear** as appropriate.

#include <comdef.h>

[Compiler COM Support Class Overview](#)

Construction

_variant_t	Constructs a <u>_variant_t</u> object.
----------------------------	---

Operations

Attach	Attaches a VARIANT object into the <u>_variant_t</u> object.
Clear	Clears the encapsulated VARIANT object.
ChangeType	Changes the type of the <u>_variant_t</u> object to the indicated VARTYPE .
Detach	Detaches the encapsulated VARIANT object from this <u>_variant_t</u> object.
SetString	Assigns a string to this <u>_variant_t</u> object.

Operators

operator =	Assigns a new value to an existing <u>_variant_t</u> object.
operator ==, !=	Compare two <u>_variant_t</u> objects for equality or inequality.
Extractors	Extract data from the encapsulated VARIANT object.

`_com_ptr_t`

Microsoft Specific

A `_com_ptr_t` object encapsulates a COM interface pointer and is called a “smart” pointer. This template class manages resource allocation and deallocation, via function calls to the **IUnknown** member functions: **QueryInterface**, **AddRef**, and **Release**.

A smart pointer is usually referenced by the typedef definition provided by the `_COM_SMARTPTR_TYPEDEF` macro. This macro takes an interface name and the IID, and declares a specialization of `_com_ptr_t` with the name of the interface plus a suffix of **Ptr**. For example,

```
_COM_SMARTPTR_TYPEDEF( IMyInterface, __uuidof( IMyInterface ) );
```

declares the `_com_ptr_t` specialization **IMyInterfacePtr**.

A set of [function templates](#), not members of this template class, support comparisons with a smart pointer on the right-hand side of the comparison operator.

#include <comdef.h>

[Compiler COM Support Class Overview](#)

Construction

_com_ptr_t	Constructs a <code>_com_ptr_t</code> object.
----------------------------	--

Low-level Operations

AddRef	Calls the AddRef member function of IUnknown on the encapsulated interface pointer.
Attach	Encapsulates a raw interface pointer of this smart pointer’s type.
CreateInstance	Creates a new instance of an object given a CLSID or ProgID .
Detach	Extracts and returns the encapsulated interface pointer.
GetActiveObject	Attaches to an existing instance of an object given a CLSID or ProgID .
GetInterfacePtr	Returns the encapsulated interface pointer.
QueryInterface	Calls the QueryInterface member function of IUnknown on the encapsulated interface pointer.
Release	Calls the Release member function of IUnknown on the encapsulated interface pointer.

Operators

COM

operator =	Assigns a new value to an existing _com_ptr_t object.
operators ==, !=, <, >, <=, >=	Compare the smart pointer object to another smart pointer, raw interface pointer, or NULL .
Extractors	Extract the encapsulated COM interface pointer.

_com_error

Microsoft Specific

A **_com_error** object represents an exception condition detected by the error-handling wrapper functions in the header files generated from the type library or by one of the COM support classes. The **_com_error** class encapsulates the **HRESULT** error code and any associated **IErrorInfo** object.

#include <comdef.h>

[Compiler COM Support Class Overview](#)

Construction

_com_error	Constructs a <u>_com_error</u> object.
----------------------------	---

Operators

operator =	Assigns an existing <u>_com_error</u> object to another.
----------------------------	---

Extractor Functions

Error	Retrieves the HRESULT passed to the constructor.
ErrorInfo	Retrieves the IErrorInfo object passed to the constructor.
WCode	Retrieves the 16-bit error code mapped into the encapsulated HRESULT .

IErrorInfo functions

Description	Calls IErrorInfo::GetDescription function.
HelpContext	Calls IErrorInfo::GetHelpContext function.
HelpFile	Calls IErrorInfo::GetHelpFile function
Source	Calls IErrorInfo::GetSource function.
GUID	Calls IErrorInfo::GetGUID function.

Format Message Extractor

ErrorMessage	Retrieves the string message for HRESULT stored in the _com_error object.
------------------------------	--

ExepInfo.wCode to HRESULT Mappers

HRESULTToWCode	Maps 32-bit HRESULT to 16-bit wCode .
WCodeToHRESULT	Maps 16-bit wCode to 32-bit HRESULT .

MIDL Predefined and Base Types

MIDL supports the following base and predefined types.

<i>Data type</i>	<i>Description</i>	<i>Default sign</i>
Boolean	8 bits. Not compatible with oleautomation interfaces; use VARIANT_BOOL instead.	Unsigned
byte	8 bits.	(not applicable)
char	8 bits.	Unsigned
double	64-bit floating point number.	(not applicable)
error_status_t	32-bit unsigned integer for returning status values for error handling.	Unsigned
float	32-bit floating point number.	(not applicable)
handle_t	Primitive handle type for binding.	(not applicable)
hyper	64-bit integer.	Signed
int	32-bit integer. On 16-bit platforms, cannot appear in remote functions without a size qualifier such as short , small , long or hyper .	Signed
__int32	32-bit integer. Equivalent to long .	
__int3264	An integer that is 32-bit on 32-bit platforms, and is 64-bit on 64-bit platforms.	Signed
__int64	64-bit integer. Equivalent to hyper .	
long	32-bit integer.	Signed
short	16-bit integer.	Signed
small	8-bit integer.	Signed
void	Indicates that the procedure does not return a value.	(not applicable)
void *	32-bit pointer for context handles only.	(not applicable)
wchar_t	16-bit predefined type for wide characters.	Unsigned

fim.