Ambientes de Desenvolvimento Avançados

http://www.dei.isep.ipp.pt/~jtavares/ADAV/ADAV.htm

Aula 14

Engenharia Informática

2006/2007

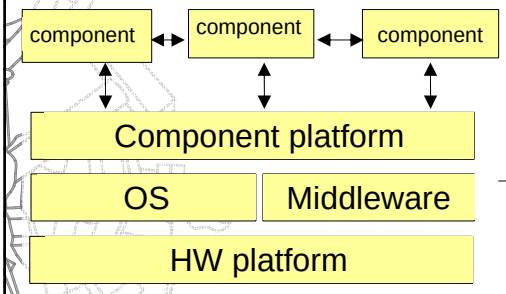
José António Tavares
jrt@isep.ipp.pt

COM

Agenda

- **Introduction to COM**

Architecture of Component Models



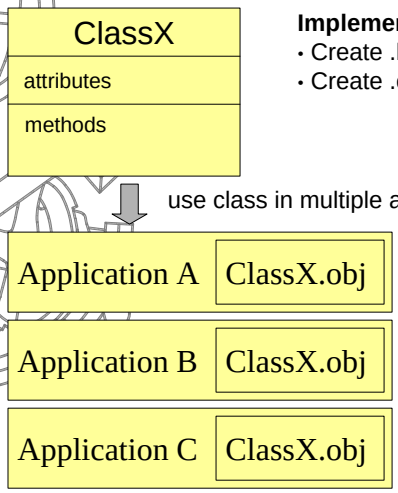
Platform Service
- binding
  - locating components
  - linking components
  - replacing components

COM (Component Object Model)

What is COM?

An architecture for component reuse that allows dynamic and efficient composition of systems from independently developed binary components.

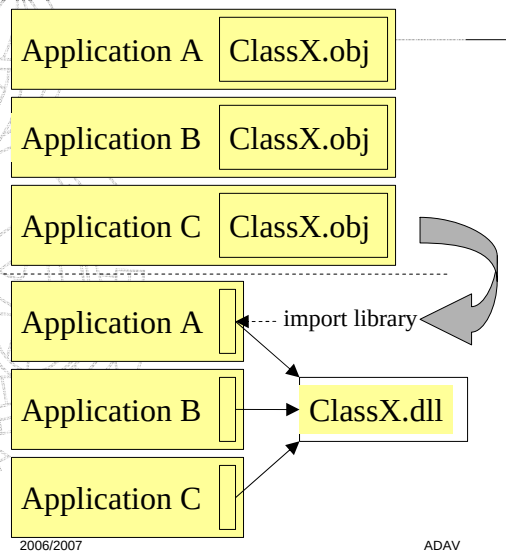
Software Distribution



Implement class in C++:
- Create .h file
- Create .cpp file

Disadvantages:
- Duplication of binary code (on disk and memory)
- Update of class X requires rebuild (even if interface does not change)

Dynamic Linking



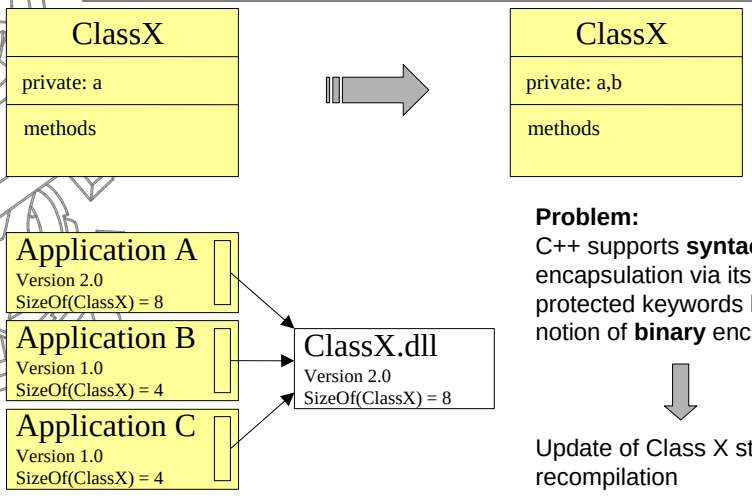
DLL is binary code that is linked to the application at run-time.

To use a DLL you will have to add an import library to your project.

Problems were:
- Duplication of code
- Update requires recompilation

Problems solved:
- Duplication of code

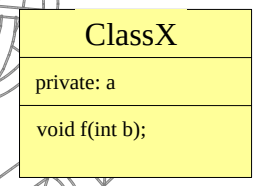
Encapsulation



Problem:
C++ supports **syntactic** encapsulation via its private and protected keywords but has no notion of **binary** encapsulation**.**

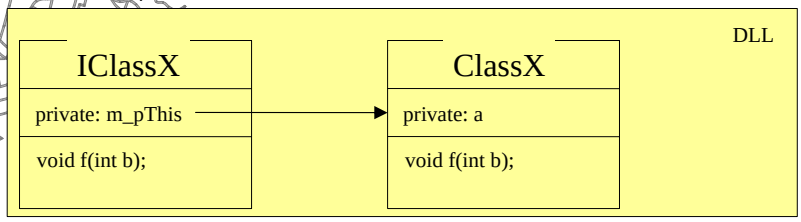
Update of Class X still requires recompilation

Separating Interface from Implementation



The binary layout of the interface does not change as data members are added to or removed from implementation class.
The interface class will become the only entry point into the DLL and it's binary signature will never change.

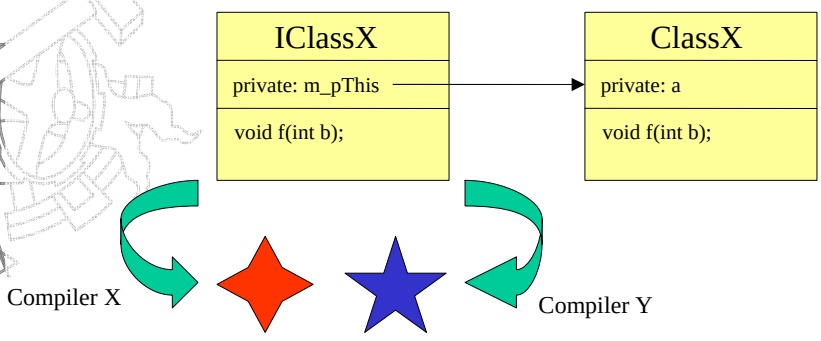
Solved problems:
•update of class X requires recompilation



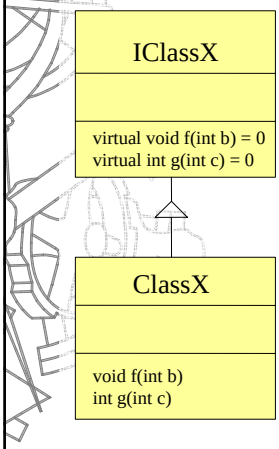
Abstract Bases as Binary Interfaces

The interface class will become the only entry point into the DLL and it's binary signature will never change.

Problem:
The binary signature is compiler dependent !



Abstract Bases as Binary Interfaces



Problem:

- Binary interface can be compiler dependent

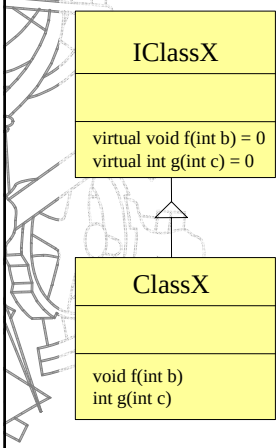
Solution:

The binary firewall imposed by the interface class can not use compiler-variant language features.

- No data members (attributes)
- Interface members are pure virtual functions

Assumption:
All compilers on a given platform produce equivalent machine code for the virtual function call mechanism !!!

Abstract Bases as Binary Interfaces



To reinforce this, it is useful to define the interface members as pure virtual functions.

- The generated machine code doesn't need to be identical for all compilers, it needs to be equivalent.
- This means that each compiler must make the same assumptions about how an object of such a class will be represented in memory and how its virtual functions are dynamically invoked at runtime.
- It turns out that in almost all compilers virtual functions take the form of virtual function pointers and virtual function tables.

Runtime Polymorphism

Because the client never needs to link against the DLL's import library, the client has no load dependences on the DLL.

This means:

- The client can execute on machine that does not have DLL installed.
- The client can load DLL on demand
- The client can dynamically select between various implementations.

Object Extensibility

The techniques presented so far allow clients:

- Select and load binary components dynamically. This means their implementation can evolve over time without recompilation of the client.

However:

- The interface of an object cannot evolve over time, because of the compilation against the signature of the interface class.

Despite the immutability of interfaces, it is often necessary to expose additional functionality !

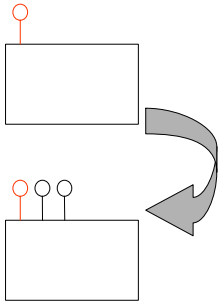
Object Extensibility

Problem:

Interfaces must be immutable and cannot change once published.

Solution:

Allow an implementation class to expose more than one interface.



The client should be able to determine at runtime whether requested functionality is indeed supported by the object currently in use. (dynamic_cast)

Resource Management

The use of dynamic_casts usually results in duplication of pointers and in complex code it is hard to keep track of these pointers.

Problem:

It's hard to keep track off the number of references to an object (needed to manage the lifetime).

Solution:

We allow object to manage its own lifetime !

What have we done?

Simple C++ class → Reusable binary component

- Deploy in binary format
- Separate interfaces from implementations
- Use C++ compiler independent interfaces
- Introduce techniques for dynamically selecting implementations
- Use construct for dynamically discovering whether object implements desired interface.

→ **We have (almost) described the Component Object Model !**

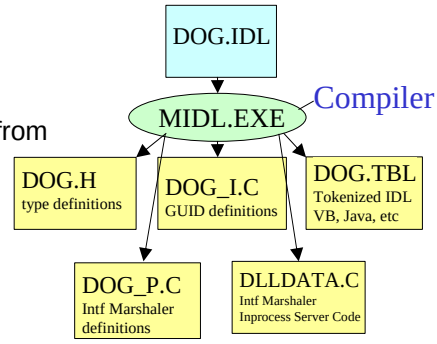
COM: Interface Description Language

We used Abstract Bases to decouple DLL from particular C++ compiler.

Client must use C++ compiler !

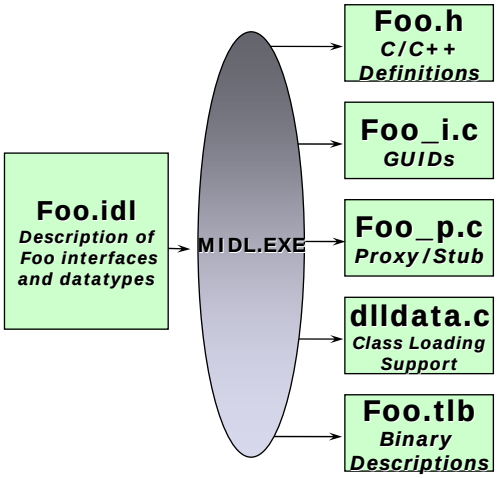
Decouple definition of interfaces from Implementation language.

COM provides Interface Description Language (IDL)

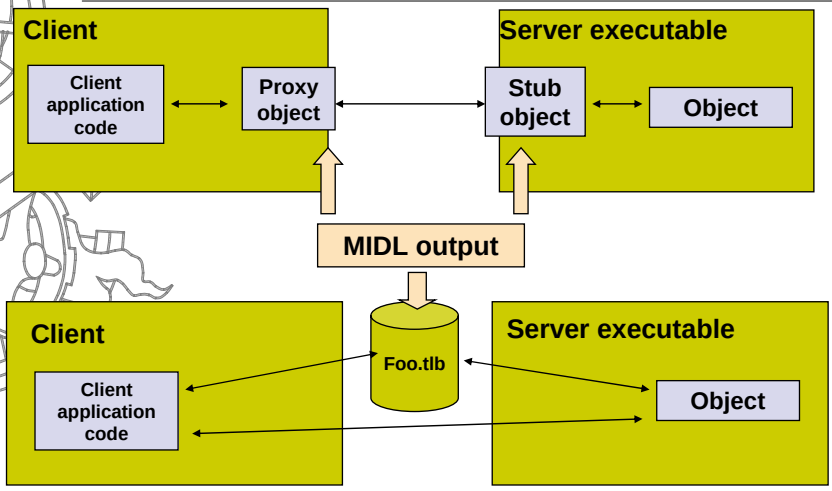


MIDL Compiler

- Microsoft IDL compiler
- Takes an IDL-file as input and generates mostly C/C++ code (not binary code) into:
  - The headers for the interfaces
  - A .tlb type library for the server (a binary)
  - The marshalling source code



MIDL Compiler, cont



COM: IUnknown

I described the need for runtime type discovery. C++ provides a reasonable mechanism (dynamic_cast).

COM also provides such a method: `QueryInterface`

`QueryInterface` is part of the `IUnknown` interface which is the root of all COM interfaces (every interface must derive from `IUnknown` directly or indirectly).

`IUnknown` also provides methods for reference counting (used to manage object life time)

COM: Activation (and SCM)

Clients need a mechanism for finding objects. Because of the dynamic nature of COM, this may involve loading a DLL or starting a server process.

There are multiple activation models. Each of the models use the COM Service Control Manager (SCM).

Each host machine supporting COM has its own SCM used only to activate the object and bind the initial interface pointer.

COM API functions: `CoGetInstanceFromFile,`
`CoCreateInstanceEx, CoGetClassObject`

COM: Classes and Servers

COM components are called COM classes and these classes are packaged in binary files called COM servers.

This binary file contains the method code for one or more COM classes. This files can be of the following types:

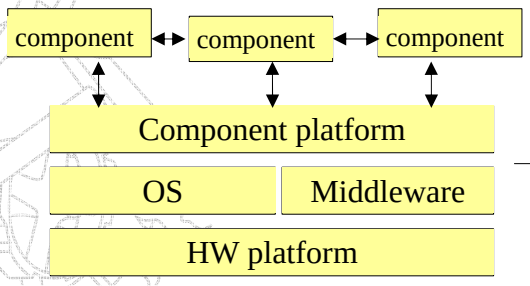
- DLL
- Normal executable

COM keeps a configuration database that maps Classes onto Servers. A local cache of the this database is called the Registry. The Win32 SDK includes a utility (REGSVR32.exe), that will install or uninstall a COM in-process server.

Properties of COM

Control Flow:	Depends on Client and COM classes (appl.specific).
Distribution:	Component can reside on multiple locations
Topology:	Dynamic (component creation/deletion/replacement)
Interaction Style:	Communications use RPCs
Binding Time:	Binding at Run Time
Binding Type:	External binding (Client)
Multiplicity:	Multiple occurrences of a COM class can exist.

Component Models: COM



DCOM
- component support is built into OS
- includes some networking services

References

Essential COM (1998)
Don Box
ISBN: 0-201-63446-5

Inside COM (1997)
Dale Rogerson
ISBN: 1-57231-349-8

GUID = Globally Unique Identifier

Ex:

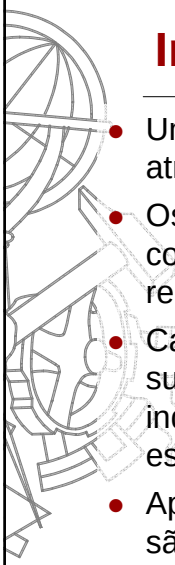
```
3fad3020-16b7-11ce-80eb-00aa003d7352
```

Estrutura de um GUID:

```
typedef struct GUID
{
    DWORD Data1;
    WORD  Data2;
    WORD  Data3;
    BYTE  Data4[8];
} GUID;

typedef GUID CLSID; /* Classe ID */
typedef GUID IID;   /* Interface ID */
```

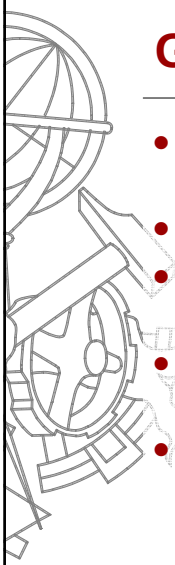

CLSID

- Each COM class is identified by a **`CLSID`**, a unique 128-bit **`GUID`** (Global Unique Identifier), which the server must register
- COM uses this **`CLSID`**, at the request of a client, to access the DLL or EXE containing the code that implements the class, which then creates an instance of the object (or finds an existing instance)



Interfaces

- Um objecto tem uma identidade e pode ser utilizado através dos seus interfaces.
- Os interfaces de um objecto não são mais do que conjuntos de funções que estão semanticamente relacionadas.
- Cada interface de um objecto significa que este suporta um conjunto de funcionalidades independentes do tempo e do espaço (através da especificação do seu **`IID`**).
- Apesar de um objecto pertencer a uma classe (**`CLSID`**) são os seus interfaces que o tornam útil…

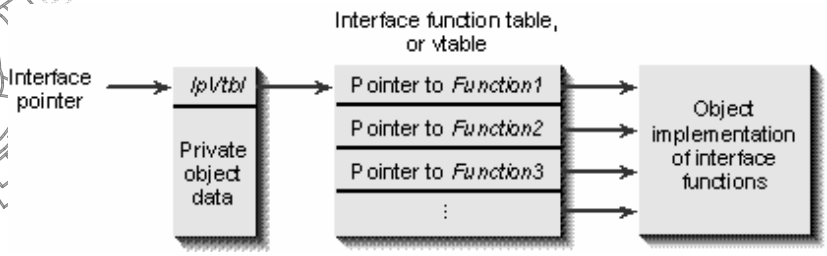


Globally Unique Interface ID

- Each interface is referred to at run time with a globally unique interface identifier (**`IID`**)
- An **`IID`** is a 128-bit number
- This **`IID`** allows a client to ask an object precisely whether it supports the interface
- Eliminates the possibility of duplication that could occur with any other naming scheme
- Permits multiple versions of the same interface with the same name

Interfaces

- A representação binária de um interface…



Interfaces

IUnknown

- O Interface **IUnknown** permite inquirir o objecto sobre outros interfaces que este suporte assim como controlar a utilização do objecto.
- Todos os Interfaces COM derivam directa ou indirectamente do Interface **IUnknown**.
- Assim, em qualquer Interface os primeiros métodos são sempre os métodos de **IUnknown**.

Interfaces

- Os métodos do interface **`IUnknown`** são:
  - **`QueryInterface`**
    – Permite obter outros interfaces suportados pelo objecto
  - **`AddRef`**
    – Incrementa o contador de referências
  - **`Release`**
    – Decrementa o contador de referências

Interface

Propriedades do método `QueryInterface`

Supondo os interfaces `IPrimeiro`, `ISegundo` e `ITerceiro`

Reflexiva	**`pIPrimeiro->QueryInterface(IID_IPrimeiro)`** deve ter sempre sucesso.
Simétrica	Se **`pISegundo`** foi obtido através de **`pIPrimeiro->QueryInterface(IID_ISegundo)`**, então **`pISegundo->QueryInterface(IID_IPrimeiro)`** tem que ter sucesso.
Transitiva	Se **`pISegundo`** foi obtido através de **`pIPrimeiro->QueryInterface(IID_ISegundo)`** e **`pITerceiro`** foi obtido através de **`pISegundo->QueryInterface(IID_ITerceiro)`**, então **`pITerceiro->QueryInterface(IID_IPrimeiro)`** deve ter sempre sucesso.

Qual o 'aspecto' de um Interface?

- Por exemplo o **IUnknown** em C++,

```
#define interface struct

interface IUnknown
{
public:
    virtual HRESULT __stdcall QueryInterface(
        /*[in]*/ REFIID riid,
        /*[out]*/ void __RPC_FAR *__RPC_FAR *ppvObject) = 0;
    virtual ULONG __stdcall AddRef(void) = 0;
    virtual ULONG __stdcall Release(void) = 0;
};
```

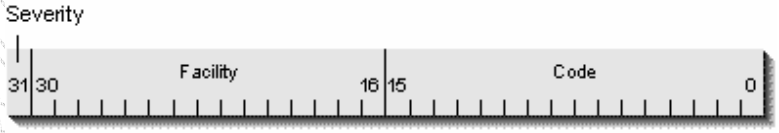

Qual o 'aspecto' de um Interface?

- 'Retirando' as especificidades de especificação do interface em C++ obtemos…

```
interface IUnknown
{
    HRESULT QueryInterface(IID& iid, void **ppv);
    ULONG AddRef(void);
    ULONG Release(void);
};
```

Qual o 'aspecto' de um Interface?

- O Retorno de quase todos os métodos de interfaces COM é um **HRESULT** (*handle to result*)…



Severity: (1 bit) Severity field
0 *Success*. The function was successful.
1 *Error*. The function failed due to an error condition.

Facility: (15 bits) Indicates which group of status codes this belongs to.
Microsoft reserves the exclusive right to define facility codes.
FACILITY_ITF is used for all errors arising from custom interfaces.

Code: (16 bits) Describes what actually took place, error or otherwise.

IUnknown and QueryInterface

```
HRESULT QueryInterface(GUID iid, void** ipptr) // referencing earlier slide
{
   if(iid == IID_ICat)    // Input GUID compared to Cat interface's GUID
        *ipptr = static_cast<ICat>(this);
   else if(iid == IID_IDog)
        *ipptr = static_cast<IDog>(this);
   else if(iid == IID_IUnknown) //Common base interface. Cat or dog?-ambiguity!
        *ipptr = static_cast<IDog>(this);
              // Implementer must choose! We choose dog
   . . .      // Other supported interfaces are checked against …
   else {     // Unsupported interface was asked for
      *ipptr = 0;
      return E_NOINTERFACE;
   }
   //  If *ipptr is non-null AddRef( ) must be called
   reinterpret_cast<IUnknown*>(*ipptr)->Addref();
   return S_OK;
}
```

Interfaces and IDL

```
[
   uuid(E4DF78F3-FE5B-11D3-9015-00C04FC12D77),
   helpstring("My Datatypes Lib")
]
library CalcTypes {    // keyword causing tlb generation
   importlib("stdole32.tlb"); // required.
   [                          // intf defined "inside lib"
      uuid(E4DF78F4-FE5B-11D3-9015-00C04FC12D77),
      object
   ]
   interface IMyInsideTypelibDefinedIf : IUnknown {
      HRESULT Method1();
   }
   [  uuid(E4DF78F5-FE5B-11D3-9015-00C04FC12D77) ]
   coclass Calc {      // reference the desired interfaces here
         [default] interface ICalculator;  // cause TLB inclusion
         interface IMyInsideTypelibDefinedIf´;
   }
}
```


Interfaces and IDL

ShopCOM IDL

Polimorfismo no COM

Objectos diferentes (de diferentes classes) podem partilhar os mesmos interfaces. Cada objecto pode implementar os interfaces de forma diferente.

Exemplo:

```
interface IAnimal : IUnknown
{
        HRESULT Eat(...);
        HRESULT Sleep(...);
        HRESULT Procreate(...);
}
```


Polimorfismo no COM

Para criar novos interfaces poderíamos fazer…

```
interface IRabbit : IAnimal
{
      HRESULT RaidGardens(...);
      HRESULT Hop(...);
      HRESULT DigWarrens(...);
}

interface IKoala : IAnimal
{
      HRESULT ClimbEucalyptusTrees(...);
      HRESULT PouchOpensDown(...);
      HRESULT SleepForHoursAfterEating(...);
}
```

Polimorfismo no COM

Ou poderíamos fazer…

```
interface IRabbit : IUnknown
{
   HRESULT RaidGardens(...);
   HRESULT Hop(...);
   HRESULT DigWarrens(...);
}

interface IKoala : IUnknown
{
   HRESULT ClimbEucalyptusTrees(...);
   HRESULT PouchOpensDown(...);
   HRESULT SleepForHoursAfterEating(...);
}
```


Polimorfismo no COM

Um objecto que implemente `IAnimal` e `IRabbit` de forma separada é funcionalmente equivalente a um que implemente o interface `IRabbit` derivado do `IAnimal`.

Implementação de Objectos

Vamos supor um objecto que enumera rectângulos…

A definição do seu interface em C++ poderia ser…

```
typedef IEnumRECT *PENUMRECT;

struct IEnumRECT
{
    STDMETHOD(QueryInterface)(REFIID, PPVOID)=0;
    STDMETHOD_(ULONG,AddRef)(void)=0;
    STDMETHOD_(ULONG,Release)(void)=0;
    STDMETHOD(Next)(DWORD, LPRECT, LPDWORD)=0;
    STDMETHOD(Skip)(DWORD)=0;
    STDMETHOD(Reset)(void)=0;
    STDMETHOD(Clone)(PENUMRECT *)=0;
};
```


A implementação de um objecto em C++ que suportasse a interface…

```
class CEnumRect : public IEnumRECT
{
private:
    DWORD m_cRef; //Reference count
    DWORD m_iCur; //Current enum position
    RECT m_rgrc[CRECTS]; //RECTS we enumerate

public:
    CEnumRect(void);
    ~CEnumRect(void);

    STDMETHODIMP QueryInterface(REFIID, PPVOID);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);

    //IEnumRECT members
    STDMETHODIMP Next(ULONG, LPRECT, ULONG *);
    STDMETHODIMP Skip(ULONG);
    STDMETHODIMP Reset(void);
    STDMETHODIMP Clone(PENUMRECT *);
};
```

A implementação do construtor

```
CEnumRect::CEnumRect(void)
{
    UINT i;

    //Initialize array of rectangles.
    for (i=0; i < CRECTS; i++)
        SetRect(&m_rgrc[i], i, i*2, i*3, i*4);

    //Ref counts always start at 0.
    m_cRef=0;

    //Current pointer is first element.
    m_iCur=0;

    return;
}
```


A implementação do método QueryInterface

```
STDMETHODIMP CEnumRect::QueryInterface(REFIID riid,
                                       PPVOID ppv)
{
    *ppv=NULL;

    if (IID_IUnknown==riid || IID_IEnumRECT==riid)
        *ppv=this;

    if (NULL==*ppv)
        return ResultFromScode(E_NOINTERFACE);

    ((LPUNKNOWN)*ppv)->AddRef();
    return NOERROR;
}
```

A implementação dos métodos AddRef e Release

```
STDMETHODIMP_(ULONG) CEnumRect::AddRef(void)
{
    return ++m_cRef;
}

STDMETHODIMP_(ULONG) CEnumRect::Release(void)
{
    if (0!=--m_cRef)
        return m_cRef;

    delete this;
    return 0;
}
```


A utilização do objecto…

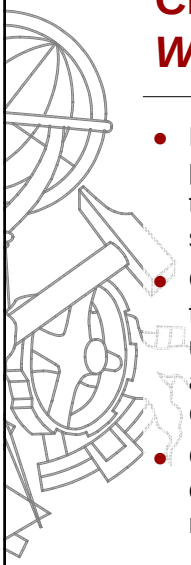
```
BOOL CreateRectEnumeratorCPP(PENUMRECT *ppEnum)
{
    PCEnumRect pER;
    HRESULT hr;

    if (NULL==ppEnum)
        return FALSE;

    //Create object.
    pER=new CEnumRect();

    if (NULL==pER)
        return FALSE;

    //Get interface, which calls AddRef.
    hr=pER->QueryInterface(IID_IEnumRECT, (void **)ppEnum);
    return SUCCEEDED(hr);
}
```

IDispatch

- 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.

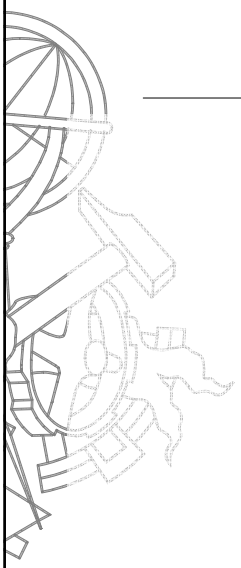
IDispatch

- Métodos do **`IDispatch`**
  - **`GetTypeInfoCount`**
    Verifica se o objecto disponibiliza informação em *runtime* (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 *`Invoke`*.
  - **`Invoke`**
    Permite aceder a métodos e propriedades disponibilizadas pelo objecto.



Criar um Servidor COM utilizando o *Wizard-ATL*

- Em termos práticos usa-se o *Wizard-ATL* do Visual Studio para a criação de um servidor COM. Isto permite esconder todos os detalhes de baixo nível da implementação de um servidor COM.
- O exemplo do servidor a criar 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.
- O *Wizard ATL* permite que sejam seleccionadas todas as operações básicas para o servidor e fazer a geração da maior parte do código necessário.



Questões

?