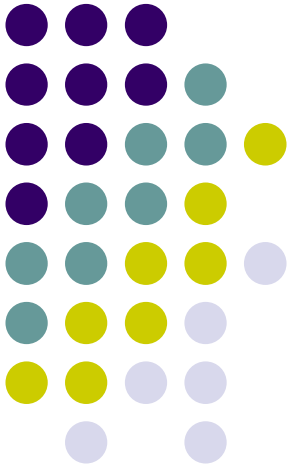
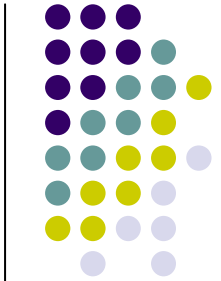
Windows - обекти и интерфейси

Интерфейсите се идентифицират по номер – 128 битово число (IID –интерфейсен идентификатор). Uuidgen – от команден ред и Guidgen от средата.
IID се генерира и с API функция CoCreateGuid().

Основен е въпросът как да изработим първи указател към наличен WO?
Възможните начина са:

1. Чрез API функции, създаващи обекти от определен тип - те връщат указател към обекта или негов интерфейс.
2. Чрез API функции, създаващи обекти, базирани на указан клас-идентификатор, Те също връщат указател.
3. Чрез методи интерфейс, връщащи указател към друг интерфейс или друг обект.
4. Чрез реализирани интерфейсни функции, които изискват другите обекти при обръщение да подават указатели към свои интерфейс





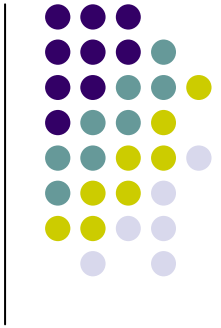
Разликите м/ду WO и C++ обектите са в:

- декларирането на класа: WO се дефинира чрез интерфейсите, които поддържа
- проявлението на обекта: class factory (с действие, подобно на new). Обектът class factory се изработва чрез специални OLE функции и поддържа интерфейс IClassFactory.
В този интефейс има функция CreateInstance() на която се подава идентификатора на интефейса от който се нуждаем.

IClassFactory::CreateInstance() == new

```
IMyInt* pMy;
CoCreateInstance(CLSID_MyObject, NULL, … , IID_IMyInt,   (void**) &pMy);
// създава обект чрез class factory и изработва IID чрез  метод на IClassFactory

…
pMy→func(…);                    //вика се функция на интерфейса.
```



обръщение към обект:

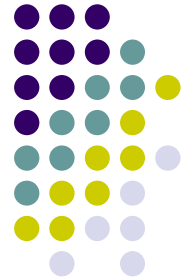
WO са достъпни чрез указател, но единствено към интерфейс.
Обръщението към member функции е тривиално:

pObject -> MemberFunction([параметри])
(За нас, всъщност, pObject е указател към даден интерфейс)

унищожаване на COM обект – метод Release().

```
ULONG __stdcall CComClass::AddRef()      //обикновено обектът, а не
                                         // клиентът вика AddRef(), когато
                                         // подава указател към интерфейс
                                         // Клиент вика тази функция само
                                         // ако копира указателя в друг.
  { return ++m_IRef; }

ULONG __stdcall CComClass::Release()
  {                                      // Release се вика от клиента
    if( __m_IRef == 0)
     {
     delete this;
     return 0;
     }
  return m_IRef;
  }
```



Общите принципи на поддържане на брояча на интерфейсни обръщения са:

1. Всяко създаване на нов интерфейс се съпътства с обръщение към AddRef()
2. Унищожаване на интерфейсен указател се съпътства с Release().
И двете функции се отнасят за целия WO, а не за определен интерфейс.

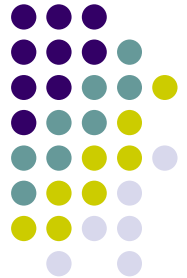
Пример:

```
LPSOMEINTERFACE pISome1;
LPSOMEINTRFACE pISome2;
CreateSomeObject( &pISome1);
.............
LPSOMEINTEFACE pCopy;
pCopy = pISome1;
pCopy->AddRef();      //броячът става 2
...............
pCopy->Release();     // броячът става 1
pCopy = pISome2;      // PCopy може да се преинициализира  след Release()
                      //едва ако броячът стане равен на 0 обекта се унищожава.
```

Изобщо броенето се подчинява на следните правила

1. Всяка функция, изработваща указател към интерфейс, трябва да се обръща към AddRef() преди края си.
2. Всяка функция се обръща към Release() преди да припокрие с нова стойност указателя . Викащите я функции се обръщат към AddRef() за предавания указател (ако има такъв), ако искат да поддържат отделно копие на указателя.
3. Вика се AddRef() за всяко локално копие на глобален указател към интерфейс.

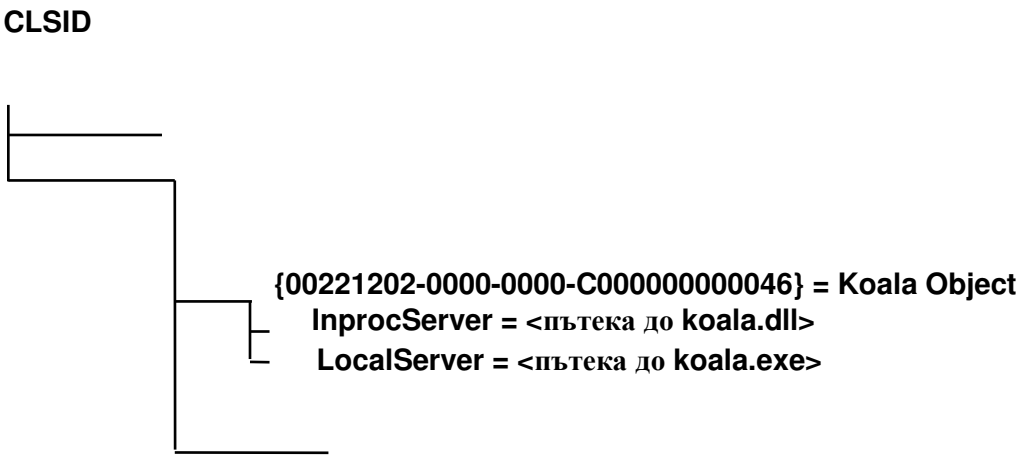
Реализация на COM обект и свързване с контейнер

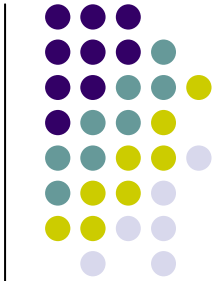


Изпълнимият файл с реализацията на COM обект се нарича COM сървър.
В секцията HKEY_CLASSES_ROOT\CLSID на регистъра на Windows има запис :

```
HKEY_CLASSES_ROOT
  CLSID
    {E312522E-A7B7-11D1-A52E-0000F8751BA7}="Foo Class"
      InprocServer32="D:\\Examples\Foo\\Foo.dll"
```

Например:





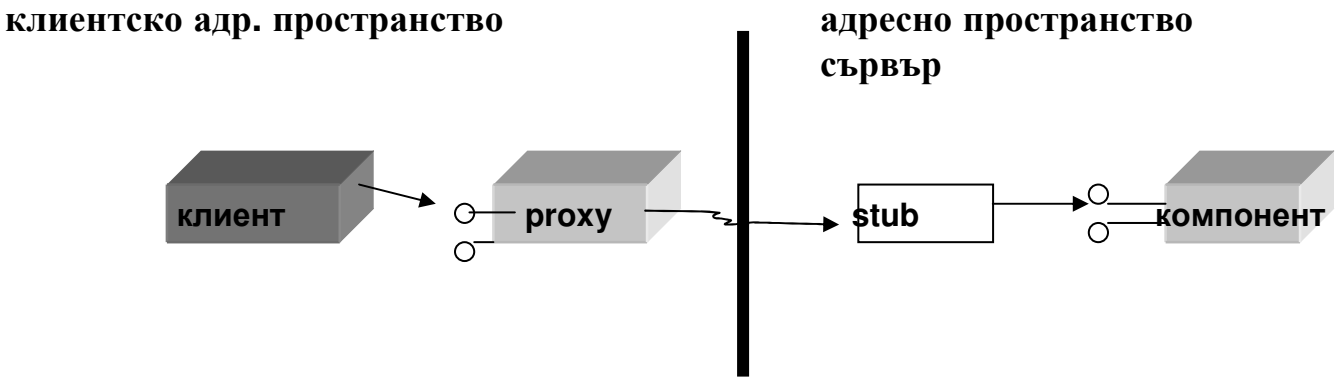
Някои понятия

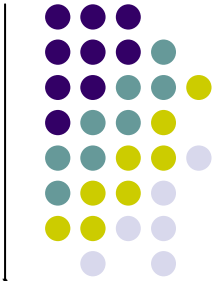
Локален сървър (in-proc) е сървърът, реализиран в DLL и работи в адресното пространство на клиента си.

Отдалечен сървър (out-of-process) е реализиран в EXE и работи в свое адресно пространство.

Хостване в DCOM

Когато сървърът е в друг процес или компютър, COM системата прави междупроцесен marchaling (вмъква съответния код за това).





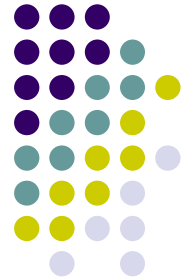
MIDL (Microsoft Interface Definition Language) компилатор, произвежда DLL с код.
Интерфейсите са описани предварително на езика IDL (Interface Definition Language).

Пример:
Ако се каним да създадем нов интерфейс IFoo2 с три метода, IDL описанието изглежда така:

```
[ uuid(E312522F-A7B7-11D1-A52E-0000F8751BA7) ]
interface IFoo2 : IUnknown
{
   HRESULT Func1();
   HRESULT Func2(int in_only);
   HRESULT Func3([in, out] int *inout);
};
```

IDL код се създава и за описание на самия обект.

```
[ uuid(E312522E-A7B7-11D1-A52E-0000F8751BA7) ]
coclass Foo
{
   [default] interface IFoo;
};
```



Интерфейс IUnknown (има IID-идентификатор – IID_IUnknown)

```
STDMETHODIMP QueryInterface(REFIID riid, void **ppvObject);
STDMETHODIMP_(ULONG) AddRef();
STDMETHODIMP_(ULONG) Release();
```

Характеристики на QueryInterface

1. **Всяко обръщение към QueryInertface за един и същ обект с цел запитване относно IUnknown винаги връща един и същ указател. Следователно за 2 интерфейса може да се определи дали принадлежат към 1 обект като се запита поотделно за IUnknown.**
2. **След като обект има проявление в паметта, интерфейсите му са статични. Това значи, че ако QueryInterface успее в една точка на програмата, то ще успее при викане на същия интерфейс във всяка друга. Това не значи, обаче, че две повиквания на 1 интерфейс ще върнат еднакъв указател. Просто интефейсът е винаги на разположение.**

3. QueryInerface() е :

рефлексивен: pInterface1->QueryInterface(IInterface1) винаги ще успее.
симетричен ако pInterface2 е изработен от
pInterface1->QueryInterface(IInterface2),
то pInterface2->QueryInterface(IInterface1) също ще успее.

транзитивен ако pInterface2 е получен чрез pInterface1->QueryIntercafe(IInterface2)
pInterface3 е получен от pInterface2->QueryInterface(IInterface2),
То повикването: pInterface3->QueryInterface(IInterface1)
ще е успешно .

4. Докато обект съществува, всички указатели към интерфейси получени от този обект са валидни, дори ако са били освободени чрез Release() тъй като броячът е общ за обект, а не за отделен интерфейс. Например:

```
    LPSOMEINTERFACE pSome;
    LPSOMEINTERFACE pOther;
    CreateSomeObject(&pSome);                    //броячът става 1
    pSome->QueryInterface(IOtherInterface, &pOther);    //броячът е 2
    pOther->Release();                                  //броячът е отново 1
    //обектът съществува, pOther е още валиден, макар и освободен
    // с Release(). Това значи, че още можем да викаме функции чрез pOther
.....
    pSome->Release();          //броячът става 0, обектът се разрушава
                               //pSome и pOther вече не са валидни
```

Нека IFoo е нашият пример за потребителски интерфейс.
За него сте дефинирали IID, например, IID_IFoo който има стойност, например :
"13C0205C-A753-11d1-A52D-0000F8751BA7" получена от генератора на GUID.
Декларацията на класа има вид:

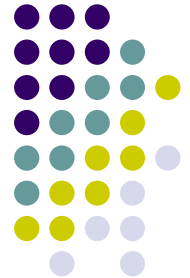
```
class IFoo
{
   virtual void Func1(void) = 0;
   virtual void Func2(int nCount) = 0;
};
```

За да стане декларацията COM-съвместима, нека леко я променим:

```
interface IFoo : IUnknown
{
virtual HRESULT STDMETHODCALLTYPE Func1(void) = 0;
virtual HRESULT STDMETHODCALLTYPE Func2(int nCount) = 0;
};
```

С използване на споменатите макроси, окончателно декларацията има вид:

```
interface IFoo : IUnknown
{
STDMETHOD Func1(void) PURE;
STDMETHOD Func2(int nCount) PURE;
};
```



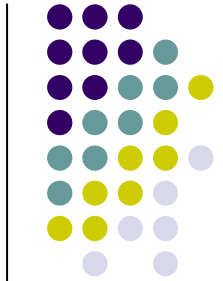
Служебната дума "interface" изгражда служебна структура. (Спомнете си, че класовете в C++ и структурите имат еднакво вътрешно представяне, с изключение на това, че структурата има public наследяване).
STDMETHOD се дефинира като __stdcall, указвайки на компилатора да генерира стандартна конвенция на повикване на функциите.
Използването на тези макроси, различно дефинирани при различните платформи, допринася за преносимостта на кода.

MFC и фабриката за класове

Какво става, когато се срещне CoCreateInstance()?
-Всяко пряко или косвено обръщение към CoCreateInstance, кара COM да претърсва регистъра за да открие CLSID на обекта, след което да може да намери реализиращия го код в съответния DLL (или EXE). CoCreateInstance всъщност извършва следното:

```
IClassFactory *pCF;
CoGetClassObject(rclsid, dwClsContext, NULL,IID_IClassFactory, (void **)&pCF);
hresult = pCF->CreateInstance(pUnkOuter, riid, ppvObj)
pCF->Release();
```

Реализация на класовия обект (Class Object)

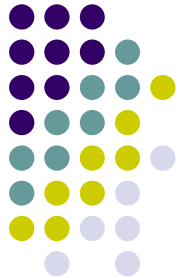


```
class CMyClassObject : public IClassFactory
{
protected:
    ULONG m_cRef;
public:
    CMyClassObject() : m_cRef(0) { };
 //IUnknown членове
       STDMETHODIMP QueryInterface(REFIID, void **);
       STDMETHODIMP_(ULONG) AddRef(void);
       STDMETHODIMP_(ULONG) Release(void);

// функции на IClassFactory
       STDMETHODIMP CreateInstance(IUnknown *, REFIID iid, void **ppv);
       STDMETHODIMP LockServer(BOOL);
};
```

Всеки COM клас трябва да поддържа своя фабрика за класове. COM фабриките за класове съдържат предимно шаблонен код, така че тяхното създаване се автоматизира лесно от развойните среди.
В MFC библиотеката е предвиден клас от тази група - COleObjectFactory.
В него са реализирани два интерфейса – IClassFactory и IClassFactory2 (използван в ActiveX технологиите), които вършат същинската работа по създаване инстанция на интересуващия ни клас.

Ето прототип на конструктора:
COleObjectFactory(REFCLSID clsid, CRuntimeClass pRuntimeClass, BOOL bMultiInstance, LPCTSTR lpszProgID);*



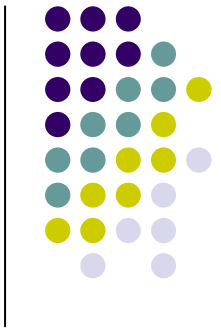
Вместо да се декларира инстанция на COleObjectFactory във всяко приложение с COM обекти, VISUAL C++ дефинира макросите
DECLARE_OLECREATE
IMPLEMENT_OLECREATE
съответно в декларацията на класа и неговата реализацията.

Най-общо да се създаде копие (instance) на COM обект има 2 начина:

1. Чрез OLE функции CoCreateInstance() или CoGetClassObject();
2. Чрез обекта - class factory и неговия метод:
IClassFactory::CreateInstance();

Ако е нужна само една инстанция на обекта, това може да стане с CoCreateInstance() с подадени CLSID и IID за искания интерфейс.

Например:

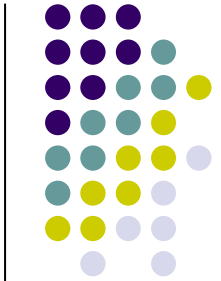


```
                ……
LPUNKNOWN pUnkOuter = NULL;
                ....
 CoCreateInstance(CLSID_Koala, pUnkOuter, dwClsCtx, IID_IPersist,
                                                (LPLPVOID) &plPersist);
```

reference към идентификатор на класа на обекта;
указател към обхващащия обект, ако обекта е създаден като част от агрегат (виж частта, описание на процеса агрегиране).

комбинация от флагове, описващи контекстав който обекта ще работи: обектът е в EXE - локален сървър, в DLL и т.н.

IID, който искаме да получим за този обект. Ако обектът не поддържа този интерфейс, функцията връща стойност за неуспех.
указател към мястото, в което CoCreate...() ще съхрани изработения указател.



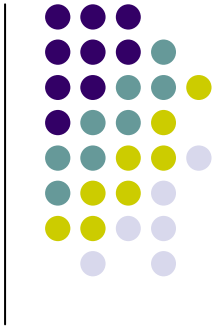
CoCreateInstance() се изпълнява на 3 стъпки:

1. **Изработва се обект - class factory и негов интерфейс - IClassFactory.**
2. **Вика се IClassFactory::CreateInstance() за създаване на нужния обект, в примера по-горе - Koala.**
3. **Вика се IClassFactory::Release() за ненужния вече обект - class factory.**

ПРИМЕР: *създаване и експониране на примерен, потребителски COM обект - "Koala" , поддържащ интерфейс IPersist:*

За целта следва да създадем два файла:–
с декларации koala.h
с описание koala.cpp
Файлът koala.h съдържа декларации на класа CKoala, интерфейс IUnknown и friend-клас: CImpPersist: public IPersist
***IPersist е реализация на стандартен интерфейс. Обектът Koala е агрегиращ (ще говорим за това по-късно).* В този клас са декларациите на експортируемите интерфейси.**

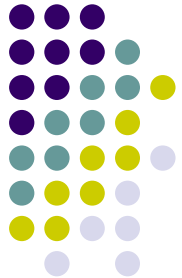
Файлът koala.cpp съдържа реализацията на класа.
Методите на обекта, които следва да бъдат реализирани са:



```
CKoala(LPUNKNOWN pUnkOuter,...);  //конструктор
~CKoala();                        // деструктор
FInit();                          // метод, изработващ указател към
                                  // IUnknown, или UnkOuther.

// трите метода на собствения за Koala COM обекта интерфейс IUnknown
QueryInterface();
AddRef();
Release();

// следват методи на вграден (агрегиран)клас със своя реализация на IPersist
CImpPersist::CImpPersist();                    // конструктор
~CImpPersist();                                // деструктор
// следват трите метода на IUnknown – собствена реализация
QueryInterface();
AddRef();
Release();
GetClassID();
```

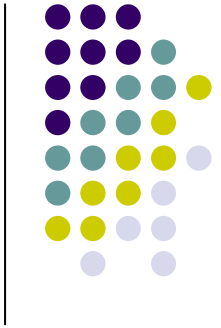


Ако обектът поддържа агрегиране, той следва да притежава средства за откриване на всички интерфейси, които са реализирани в обхващащия го клас (какъвто е случая с интерфейса IPersist на примерния обект Koala).
При агрегиране, реализацията на интерфейс изпраща повикванията на IUnknown винаги към обекта, който контролира (обхваща) настоящия.
Ето защо в конструктора на класа CKoala е предвиден указател към интерфейса IUnknown на обхващащия обект (pUnkOuter). Ако обектът "Koala" не е агрегиран (този параметър е NULL), CKoala:FInit() предава указател към собствения си IUnknown, вместо към този на обхващащия обект. Така винаги имаме достъп до интерфейсите на двата обекта- външен и агрегиран.

(Функцията FInit() е предвидена като отделна, втора фаза в конструирането на обекта "Koala". Това е направено с цел повишаване надеждността на етапа конструкция на обекта. FInit() създава CImpPersist, където поради търсене на агрегат може да има и неуспех.)

За да се експортира коректно обекта от своята реализация в сървъра:

Остава да се реализира class factory за всеки потребителски COM обект, която ще реализира създаване на инстанциите му (в случая за обекта "Koala"). Обектът Class factory поддържа стандартно интерфейс IClassFactory.
В сървъра се реализира клас CKoalaClassFactory!



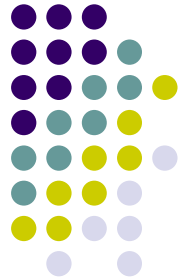
```
class _far CKoalaClassFactory : public IClassFactory
{
     protected:
     ULONG m_cRef;                    // брояч на обръщенията

     public:
     CKoalaClassFactory(void);
     ~CKoalaClassFactory(void);

     STDMETHODIMP QueryInterface(REFIID, LPLPVOID);
// STDMETHODIMP е макрос, скриващ различията в декларации
// на типове в C, C++, 16, 32 битови версии и Macintoch.
     STDMETHODIMP AddRef(void);
     STDMETHODIMP Release(void);

     STDMETHODIMP CreateInstance(LPIUNKNOWN, REFIID, LPLPVOID);
     STDMETHODIMP LockServer(BOOL);
}
```

Съществена тук е функцията CreateInstance(). Първият й параметър е указател към IUnknown на обхващащ обект (ако съществува агрегиране). Целта е да се реализира връзка с външния (обхващащ) обект и той да може да запита за IUnknown на съставния си. Ако липсва агрегиране, този параметър е NULL.



Функцията LockServer() инкрементира или декрементира брояча в сървъра. Чрез него сървърът може да остане в паметта, дори ако не обслужва повече обекти. Това е оптимизация за избягване на следващо зареждане.

Фабриката за класове трябва да се направи видима. Възможните за това реализации са различни според мястото на сървъра - в EXE или в DLL. В EXE се дефинира нова задача (task). Нека разгледаме по-простата процедура, когато се използва DLL. Всеки сървър реализира и експортира функция DllGetClassObject(), която извършва следното:

Когато потребител повика CoCreateInstance() за създаване на обект или CoGetClassObject(), COM библиотеката претърсва регистъра за запис с търсения клас- идентификатор (CLSID) за да зареди съответния сървър в паметта, след което търси в него реализация на DllGetClassObject(), извикана със заявените от потребителя параметри за CLSID и IID. Чрез тази функция се създава class factory за нужния клас идентификатор CLSID и се получава указател към заявения интерфейс. Този начален интерфейс обикновено е IClassFactory за конкретния COM обект.

Наследяемост в компонентна среда - механизми

1. съдържане (containment) - Коала съдържа Животно и реализира версия на IЖивотно , която експонира.
2. агрегиране: Коала експонира директно интерфейсите на родителя като свои. Това изисква пренасочване на повикванията към реалните родителски реализации.
Т.е следва да има специфична реализация на QueryInterface(), AddRef(), Release().
3.Старата техника на множествено наследяване
Започваме разсъждения за този метод:
искаме да създадем COM обект, поддържащ интерфейсите **IDrawing** и **IOutlet,** при това с използване на **множествено наследяване:**

```
class IOutlet :     public IUnknown {...} ;
class IDrawing :    public IUnknown {...} ;
class CLightBulb :  public IDrawing,   public IOutlet
{
  public:
   // реализация на  IUnknown
      virtual HRESULT __stdcall QueryInterface(REFIID riid, void **ppv) ;
      virtual ULONG   __stdcall AddRef() ;
      virtual ULONG   __stdcall Release() ;
   // реализация на интерфейса IDrawing
       virtual HRESULT __stdcall Draw(CDC* pDC, int x, int y);
      virtual HRESULT __stdcall SetPalette(CPalette* pPal);
      virtual HRESULT __stdcall GetRect(CRect* pRect);
   // реализация на интерфейса  IOutlet
      virtual HRESULT __stdcall On();
      virtual HRESULT __stdcall Off();
      virtual HRESULT __stdcall GetState(BOOL* pState);
…// следват детайлите по реализацията
...}
```

Доколкото в езика C++ клас и структура се третират по еднакъв начин, един интерфейс може да наследява друг, както и клас да наследява интерфейс.

Подходът с множествено наследяване има очевидни предимства: интерфейсите, които имаме намерение да имплементираме в класа просто се изброяват в списък като от базови класове. IUnknown ще се имплементира само веднаж .

дефиницията на IUnknown е еднократна, независимо от това, че той е наследяван във всеки интерфейс (и в IDrawing и в IOutlet). Това е възможно, защото чрез виртуалност може да се създаде многообразие:

```
class A
{
   virtual int f();
} ;
class B
{
   virtual int f();
} ;
class C : public A, public B
{
   virtual int f() ;
} ;
```

т.е можем да мислим за различни реализации на еднакво декларираната f() !!!

Тъй като всеки отделен интерфейс се нуждае от методите на IUnknown, при множествено наследяване може да имаме обща тяхна реализация, която повторно да използваме. Тогава, най-добре да замислим отделен клас CUnknown, който съдържа тази реализация:

```
class CUnknown : public IUnknown {...} ;
...
class CLightBulb : public CUnknown, public IDrawing, public IOutlet
      {
...
      } ;
```

За съжаление, такава реализация няма да работи както се очаква. CUnknown няма да може да имплементира интерфейса IUnknown, който искаме да е наследен и през IDrawing и IOutlet, тъй като чисто виртуални функции винаги трябва да се имплементират в (и само в) производния клас (а тук имаме допълнително и 2 родителя и наследил ги клас).

А ние искаме да имплементираме IUnknown в CLightBulb (отделно IUnknown следва да е имплементиран и в IDrawing, public IOutlet за да разпознава и техните методи. Стана объркващо!)

И така, трябва да имплементираме няколко реализации на интерфейса IUnknown:
един в CUnknown, друг в CLightBulb и по 1 в наследяваните интерфейси.

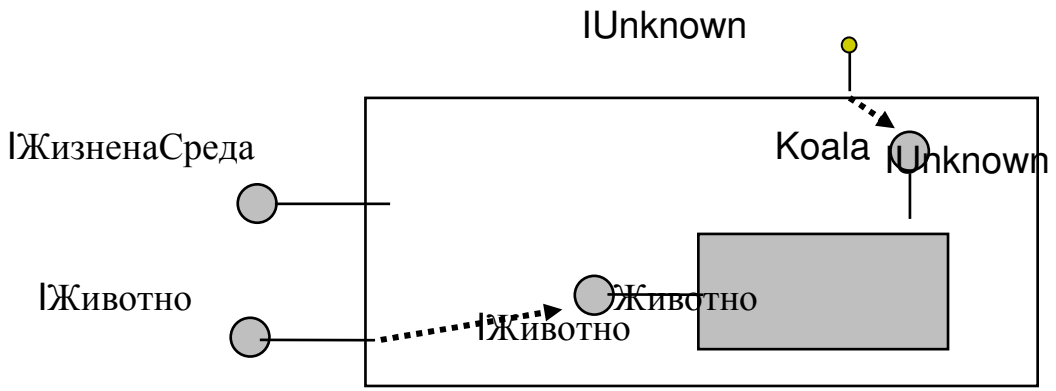
IUnknown в CUnknown ще прави това което искаме от IUnknown и
IUnknown в CLightBulb ще пренасочва повикванията към горния IUnknown.
IUnknown в CLightBulb се нарича делегиращ (пренасочващ) IUnknown, а CUnknown ще съдържа не-делегиращ IUnknown.

Преминаваме към обсъждане на механизма на **съдържане (object containment)**

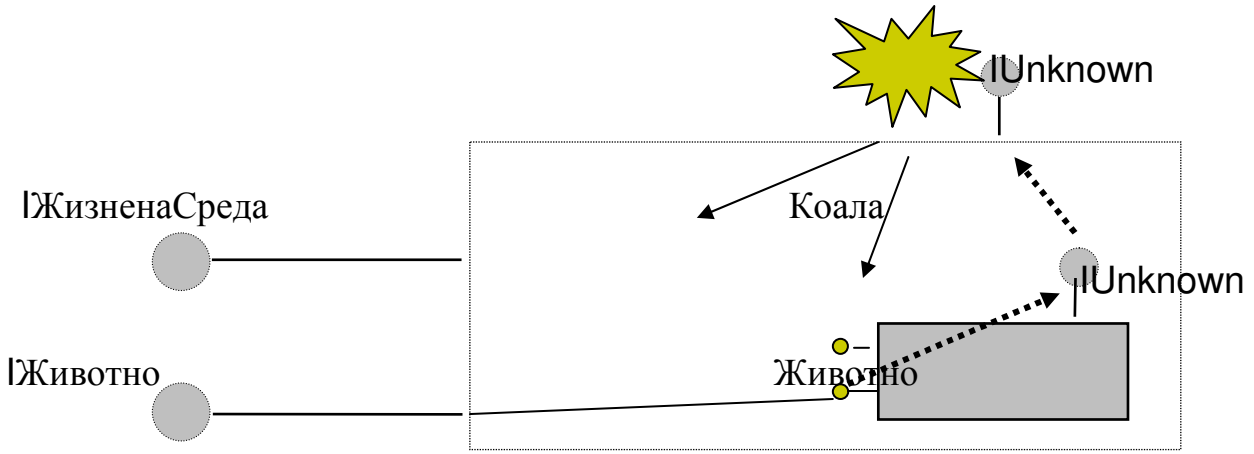
искаме да използваме съществуващ интерфейс (напр. IЖивотно) в наш компонент (CKоала). Това можем да постигнем като в нашия обект създадем инстанция (чрез CoCreateInstance()) на другия и при необходимост викаме неговите методи. Тогава нашият обект е външен, а другият е вътрешен. Ако поддържаме и подходящо сме инициализирали указател pInnerObject,сочещ към интерфейси на вътрешния обект, то с негова помощ можем да викаме методите им, като че са реализирани във външния (CKоала).

Например така:

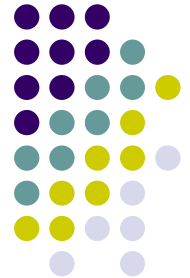
```
HRESULT __stdcall CKoala::Eating(….)
{
return pInnerObject→Eating(….);     //викаме метод Eating() на вградения компонент
                                    // от интерфейса IЖивотно
}
```



И третия подход –агрегиране (object agregation)



Пренасочването на повикванията е съществената част за разбиране на агрегацията. Тя пък е основния механизъм в компонентното програмиране



```
// CSomeObject е агрегиращ обект, реализиращ IUnknown и ISomeInterface
class CSomeObject : public IUnknown
{  private:
            DWORD          m_cRef;          //  брояч на обръщенията
            IUnknown*      m_pUnkOuter;     // Controlling  IUnknown
```

```
// вътрешен клас реализира интерфейс  ISomeInterface:
class CImpSomeInterface : public ISomeInterface
     {         friend class CSomeObject ;
         private:
           IUnknown*     m_pUnkOuter;     // контролиращо неизвестно
         public:
           CImpSomeInterface() { m_cRef = 0;    };
          ~ CImpSomeInterface(void) {};

    // IUnknown методи, които насочват  към външния IUnknown

     STDMETHODIMP    QueryInterface(REFIID riid, void** ppv)
                                    {   return m_pUnkOuter->QueryInterface(riid,ppv);  };

     ...
              //Методи на  ISomeInterface
              STDMETHODIMP SomeMethod(void)              { ...   };
      } ;
```

```
CImpSomeInterface m_ImpSomeInterface ;     //инстанция на вътрешния клас
```

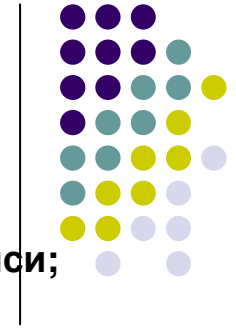
```
public:
        CSomeObject(IUnknown* pUnkOuter) // конструктор на външния клас
        {  инициализации
             ...
        } ;
        ~CSomeObject(void) {} ;            // деструктор
```

Методи
На външния

```
    static HRESULT Create (IUnknown* pUnkOuter, REFIID riid, void **ppv)
        {
      CSomeObject*      pObj;
      if (pUnkOuter != NULL && riid != IID_IUnknown)
        return CLASS_E_NOAGGREGATION;
      pObj = new CSomeObject(pUnkOuter);
...
        }

    // Методи на  IUnknown на външния клас, които не пренасочват
    STDMETHODIMP QueryInterface(REFIID riid, void** ppv)
        {
                if (riid == IID_IUnknown)       *ppv=this;
                if (riid == IID_ISomeInterface)    ....
         } ;
        STDMETHODIMP_(DWORD) AddRef(void) {…}
        STDMETHODIMP_(DWORD) Release(void) {…}
};
```

Обърнете внимание на двойното дефиниране на методи на IUnknown. Реализацията им е различна.



предимства :

- възможност за съществуване на методи с еднаква сигнатура в различни интерфейси;
-единно броене на референциите към общия, агрегиращ обект и оттук облекчен механизъм за създаване и унищобаване;
- централизирано насочване на повикванията от единен, неделегиращ QueryInterface().

Агрегация и MFC

MFC крие механизма в набор от макроси. Основните от тях са:

```
BEGIN_INTERFACE_PART( име_клас, име_на_интерфейс)
…
STDMETHOD(име_на_интерфейсен_метод)(списък от параметри);
STDMETHOD(име_на_интерфейсен_метод)(списък от параметри);
……
END_INTERFACE_PART(име_клас)
```