OLE Automation

цел :

Да имаме достъп до функционалност на приложение от скриптов език (Visual Basic, Java Script и т.н.) Скриптовите езици не поддържат COM технология и COM интерфейси , виртуални таблици (v-tables) и интерфейси.

Automation сървъри са и Excel, Microsoft Word, Office пакет, PowerPoint.

Основни понятия

COM обектът казваме,че експонира интерфейси и методи.

Automation обектът експонира методи и свойства (properties). Скрипт език може да ги използва, все едно са негови (спомненте си, че COM обектът не дава достъп до променливи).

Script езиците с Automation поддръжка притежават run-time scripting machine (V.Basic например), посредник между script интерпретатора и Automation сървъра.

Visual C++ има обвиващи класове и прави лесен hosting (лесно вграждането) на Automation сървърен обект, както ще покажем в края на темата.

Базов интерфейс е IDispatch.

Чрез него OLE Autom.сървъра експонира свои properties (данни за настройка – напр. цвят на фон, шрифт и т.н.) и методи. Клиентските програми се наричат OLE Automation контролери:

IDispatch

OLE Automation server

OLE Automation controller

обща схема на програмната среда

Интерфейсът IDistatch

Всеки OLE Automation сървър следва да реализира този интерфейс.
За да се разбере дали един COM обект поддържа OLE автоматизация, OLE контролерът следва да запита по стандартния начин с QueryInterface() за IDispatch.
Технологията е езиково независима. Следователно данните които се обменят между контролер и сървър следва да са в независим формат.

нови типове за данни

BSTR

подобен на С, като се предхожда от 4 байтово поле съдържащо дължината:

Брой байтове, не букви | UNICODE кодировка

6	a	b	c	d	e	f	\0

Windows NT предостявя алтернативни API функции за работа с BSTR:
за създаване на BSTR и попълване с низ;
за освобождаване;
за промяна;
за обработки с дължината.

MFC има също методи за преобразуване BSTR ⟷ CString, напр:

```
Cstring string = (LPCWSTR) bstr;
или:
CString str = _T("Hello");
BSTR bstr = str.AllocSysString();
```

VARIANT структурата се използва при трансформиране и предаване на различни типове данни. Всъщност това е самоописващ се тип данни:

```
typedef struct tagVARIANT VARIANT;
typedef struct tagVARIANT
{
VARTYPE vt;                    //флагове, идентифициращи типа на данните
union
{
//стойността
short iVal;  long lVal;   float fltVal;   double dblVal;   VARIANT_BOOL bool;
SCODE scode;   DATE date;   BSTR bstrVal;  IUnknown FAR * punkVal;
IDispatch FAR* pdispVal;            SAFEARRAY FAR* parray;
//полета, използвани при предаване на данни по указател
short FAR* piVal;  long FAR* plVal;   float FAR* pfltVal;
double FAR* pdblVal;   VARIANT_BOOL FAR* pbool;
DATE FAR* pdate;     BSTR FAR* pbstrVal;      IUnknown FAR* FAR* ppunkVal;
IDispatch FAR* FAR* ppdispVal;         VARIANT FAR* pvarVal;  void FAR* byref;
        };
```

За всяко от полетатата на VARIANT има съответстващ идентификатор:

VT_I4	4-byte signed int
VT_R4	4-byte real
VT_R8	8-byte real
VT_DATE	date
VT_BSTR	binary string
VT_DISPATCH	IDispatch FAR*
VT_ERROR	SCODE
VT_BOOL	True=-1, False=0
VT_UNKNOWN	IUnknown FAR*
VT_UI1	unsigned char
VT_LPSTR	null-terminated string

Има API функции за работа с VARIANTструктури (например ::VariantInit()).

При автоматизация задължително се работи с данни от представими във VARIANT типове COM управлява добре тези типове, така че не са нужни proxy / stub код в отделна DLL библиотека (както е при out of process сървърите).

VARIANT позволява предаване на всякакъв вид данни.
IDispatch използва обекти VARIANT за обмен на данни с /от OLE Automation сървъри.

чрез тази структура могат да се предават: числа без знак 1 байт, указатели към беззнакови 1 байт букви, цели, указатели към тях,4 байта цели, указатели към тях, IEEE реални и указатели към тях (4 и 8 байта), BSTR и указател към него, булеви и указатели към тях, дата и час, указател към обект поддържащ IDispatch, или към IUnknown, масив от данни тип <anything> и т.н. Предвидени са функции за работа с обект VARIANT. Те фактически работят коректно с всякакъв тип в VARIANT и извършват коректно нужните преобразувания и обработки над данните д него.

тип SAFEARRAY
Тип данни за масив използван в автоматизацията.

API има множество функции (започват с SafeArray). MFC има поддържащ клас за работа с тези структури – COleSafeArray.
Ето как е дефиниран типа SAFEARRAY :

```
typedef struct tagSAFEARRAY
   {
   USHORT    cDims;                //размерности на масива
   USHORT    fFeatures;            //Flags описващи масива
   USHORT    cbElements;           // размер на всеки елемент
   HANDLE    handle;               //HGLOBAL към масива
   void *    pvData;
   SAFEARRAYBOUND    rgsabound[]; // граница за всяка размерност
   } SAFEARRAY;

typedef struct tagSAFEARRAYBOUND
   {
   ULONG    cElements;             // брой на елементи във всеки размер
   long     LBound;                //долна граница за всяка размерност
   } SAFEARRAYBOUND
```

IDispatch

Основната функция, която вика метод или пропърти, използайки за целта идентификатори на DISPATCH (DISPID) е *IDispatch::Invoke*.

В момента, когато контролера получи указател към този интерфейс (dispinterface), то той притежава указател към реализацията на *IDispatch* с която може да има достъп до набора dispID, специфични за реализацията.

Когато в кода се повика *IDispatch::Invoke*, действителния преход към метод или пропърти се определя едва в run time.

По-долу следват методите на интерфейса:

```
interface IDispatch : public IUnknown
{
public:
HRESULT GetTypeInfoCount(….);        /* използва се да изработи броя
                                      ITypeInfo интерфейси (0 или 1)
                                       поддържани от сървъра. */
HRESULT GetTypeInfo(…);              /* използва се да изработи указател към
                                      ITypeInfo, който е нужен за определяне
                                      дали сървърът поддържа OLE Automation*/
HRESULT GetIDsOfNames(…);            /* преобразува име на метод, пропърти
                                      в DISPID (всъщност индекс към името,
                                      използван при обръщенията към IDispatch) */
HRESULT Invoke(…);                   /* същинска функция. Тя се използва за
                                      достъп до пропърти или метод
                                      експониран от OLE Automation сървъра. Чрез нея
                        се разпитва сървъра  за експонираните си интерфейси */
};
```

В автоматизацията (или чрез IDispatch) винаги се работи не с имена а с dispatch ID (dispid), които се изработват от GetIdsOfNames(), преди Invoke().

Всъщност при всяко повикване по име на метод или пропърти става следното:

1. изработва се CLSIDза обекта (от името му) – за целта има API ф-ия ::CLSIDFromProgID();
2. вика се CoCreateInstance() за създаване на обекта;
3. получава се указател към IDispatch;
4. изработват се DISPID за данновите членове с GetIDsOfNames();
5. подготвя се масив с аргументите на повикване от тип VARIANTARG;
6. обръщение към метода Invoke() с подадени параметри (указва се дали метод или property);
7. резултатът, който се получава е в тип VARIANTструктура.

Защо изобщо е необходим механизма с IDispatch , а не конвенционалния с интерфейсите. Обяснението е следното:

Скриптовете и интерпретаторните езици не поддържат работа с v-tables и интерфейси. Следователно и с COM интерфейсите.
C++ прави това като разчита интерфейсните дефиниции в заглавния файл, описващ обекта (файл с разширение .odl). Това обаче не могат скриптовете езици. Те се нуждаят от механизъм, прехвърлящ преобразуването на име на метод към указател, сочещ функцията , която го реализира.
Този механизъм се базира на универсален интерфейс IDispatch.

Следователно преобразуванията име → указател в COM се извършват от викащия, през виртуални метод таблици, инстанциирани по време на компилация, докато при автоматизацията → през индексни таблици в самия обект в run-time.

Това е тъй наречения метод на късно свързване (late binding механизъм).

Късно свързване се поддържа от скриптовите и интерпретируемите езици.
Например във Visual Basic повиквания за IDispatch има неявно винаги, когато променлива се декларира от тип Object:

```
Dim myobj as Object
set myobj = CreateObject(TV.Remote")
myobj. interface_method
```

Същото в ASP става , когато се използва COM обектът Server:

```
Dim myobj
set myobj = Server.Create("TV.Remote")
myobj.interface_method
```

Забележка:
В този механизъм не се използва файл, наречен типова библиотека (.tlb,или .odl) . В него са описани всички интерфейси, методи и пропъртита на COM обекта, както и параметрите на повикване на всеки метод. Типовата библиотека е регистрирана в регистъра, така че клиентът черпи информация от нея за COM обекта.

Типова библиотека се създава с COM API функции, или от IDL файл. Компилаторът от MIDL (Microsoft Interface Definition Language), който е час от VC++ пакета, чете такъв файл и при компилация изработва типова библиотека. VISUAL C++ създава файл ODL към проекта при указание за създаване на Automation сървър и там описва обекта и интефейсите му. Помощна програма MTypeLib компилира този файл и генерира типова библиотека.

Другият метод на свързване е ранното свързване (early binding) – типичен за C++. Там преобразуванията за достигане до указател за метод (работи се с интерфейси) се изпълняват по време на компилация.
Работи се с v-tables. Обектът може да не поддържа IDispatch, а само интерфейси.
Ако има и IDispatch, то следователно имаме двоен интерфейс (в един интерфейс има и методи на IUnknown и на IDispatch + специфичните му методи). Не се създава DISPID за достъп до методи и свойства, а се работи с виртуални таблици. Може да се работи и с IDispatch и ако са нужни DISPID, се работи с информацията от типовата библиотека. Обекти с двойни интерфейси поддържат ранно и късно свързване.

В MFC такива интерфейси се създават трудно. Препоръчва се използването на алтернативната класова йерария ATL (Active Template Library).

Има и трети начин на свързване (междинен) *– ID binding:*

Всички DISPID към свойства и методи могат да се получат от типовата библиотека с която се свързваме в compile time.Следователно не е нужно повикване на GetIDsOfNames() за всяко повикване. Имаме само 1 повикване на GetIDsOfNames(), както в early binding за изработване на CLSID за свързване с обекта (всъщност с типовата му библиотека) в compile time. Invoke() повикванията са в run time и изискват GetIDsOfNames() обръщения, но от типовата библиотека.

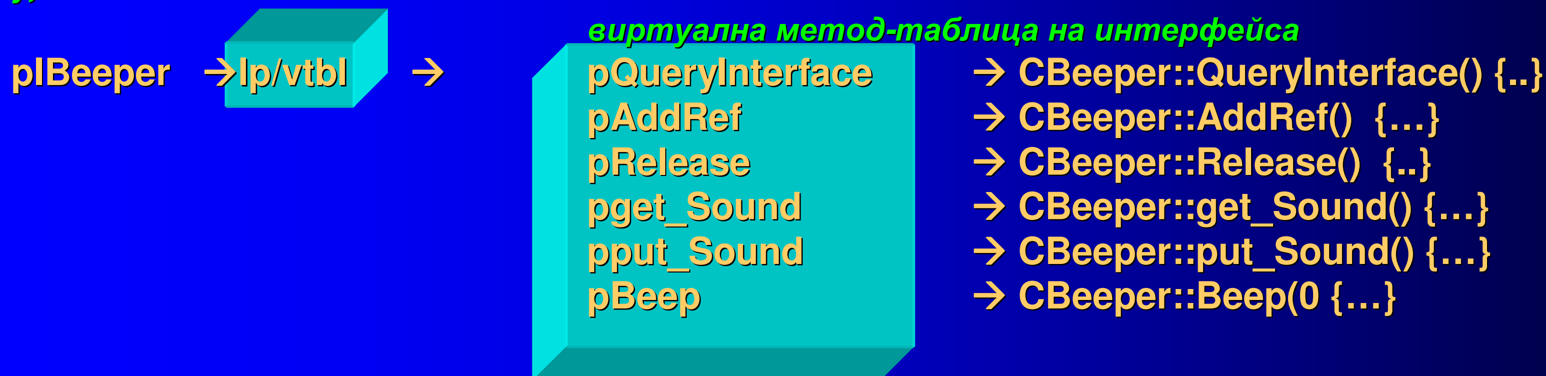
Методът е поне 2 пъти по-бърз от late binding.

Разликите между ранното и късното свързване могат да се пояснят в следния пример: Нека имаме прост обект именован Beeper (клаксон). Той притежава един даннов елемент (property) с име Beep. Реализацията на този обект нека е на C++, като експонираната му функционалност става с помощта на OLE механизмите:

```
class CBeeper
{ public:
        long        m_lSound;                  // това е данновия експониран член
public:
        long        Beep(void);                // това е метода на класа
};
```

дефинираме интерфейс, например с име IBeeper.

```
interface IBeeper : IUnknown
{
long get_Sound( void);
void put_Sound( long lSound);
long Beep( void);
};
```



ограниченията с виртуалните таблици, каквато е тази за IBeeper, са че всеки клиент на този обект следва да се привърже (binding) към интерфейсните му методи през указатели.
Т.е. ред от клиентски код с обръщение pIBeeper →Beep() например, трябва да може да се компилира в call инструкция на процесора с параметър – отместване спрямо началото на виртуалната таблица (или все едно спрямо pIBeeper).

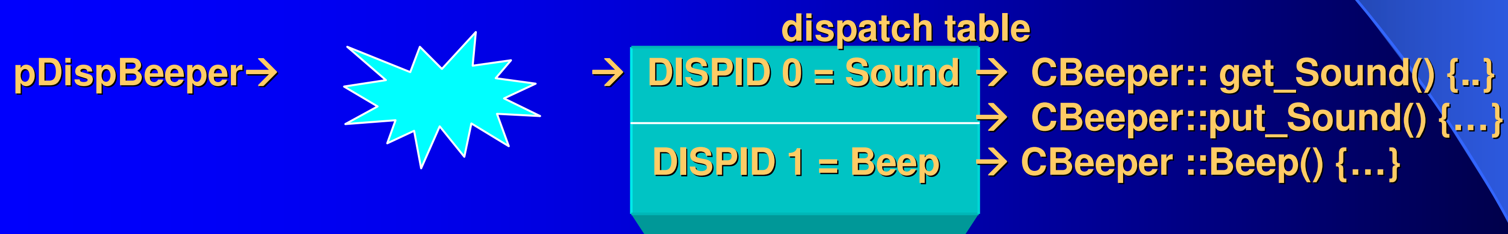
Тази техника работи лесно и бързо за компилируем код, но не и за интерпретируем.

В този случай достъп до функционалността на обекта става на основата на късното свързване (late binding) чрез посредничеството на dispinterface. В неговото описание (по-долу има пример на такова) в odl файл, пропъртита и методи се съпоставят на собствени идентификатори (заградените в прави скоби), използвани пряко за връзката. Следва дефиниция на dispinterface за нашия обект:

```
dispinterface DIBeeper
{
properties:
        [id(0)]     long Sound;
methods:
        [id(1)]     long Beep(void);
};
```

За пропъртито Sound беше съпоставен dispID нула, а за метода Beep() – dispID единица. Контролерът (клиентът) използва тези dispID в run-time за да насочва точно повикванията към методи и пропъртита към мястото на тяхната реализация в обекта:

флаг определя операция над пропъртито get или set



Изработването на интерфейсния указател към IDispatch се подчинява на общите механизми в COM (през QueryInterface). Тогава се реализира и преобразуванието от pDisp към dispatch таблицата (т.е. всичко това става както в early binding).
Късното свързване намира място едва при намиране съответствие за пропърти или метод от dispatch таблицата, което се решава в run-time.

Процедура на създаване на OLE Automation server с използване на MFC класове

В MFC поддръжката за Automation (интерфейс и специфичните типове се конструират автоматично) е предвиден клас COleDispatchImpl, чиято инстанция се реализира още в *CCmdTarget::EnableAutomation().* Т.е всеки MFC клас, поддържащ автоматизация следва да наследи CCmdTarget и да повика горната функция още в конструктора.

Преобразуванията:

име метод/свойство → DISPID

нужни за повикванията през Invoke() стават с dispatch карта:

```
BEGIN_DISPATCH_MAP (клас, родител)
        DISP_FUNCTION (клас, име на dispatch метода, име на функция с
                    която е реализиран, ID (от полетата на VARIANT) за
                    параметрите с които тя се вика)
        DISP_PROPERTY_EX (клас, име на свойството, методи Get..  Set.. на
                    класа с които се обработва свойството)
END_DISP_MAP()
```

В картата (генерирана от Class Wizard) липсват DispID. Всъщност има позиционна зависимост: място в картата → номер на DispID в ODL генерирания файл.

примерен OLE сървър (AutoBub), който изобразява в прозорец окръжности

(този OLE сървър има свое изображение).

1. В AppWizard създавате ново SDI приложение с това име. Указвате “support OLE Automation”. При диалозите на AppWizard, в поле File Type ID , въвеждате ProgID на Automation обекта.

2. избирате CAutoBubDoc (само 1 клас има поддръжка на automation).

3. правите добавки в класа, нужни за нормалната функционалност: структура с данните за отделна окръжност, функциите за добавяне, броене и извличане на окръжност (AddBubble(), GetBubble(), GetCount()) като прототипи и реализация.

4. Създава се диалогов прозорец (с ресурсен редактор) за описване параметри на нова окръжност (.rc). Задават се идентификатори на диалоговия прозорец и на отделните контроли в него.

5. Създава се CAddBubDlg, наследник на CDialog, свързан с идентификатора на ресурса на създадения диалогов прозорец.

6. добавя се меню опция "Add bubble". Свързва се с обработваща меню-опцията функция – тя очевидно е метод на CAutoBubView:

```
void CAutoBubView :: OnMenuAddBubble()
{
// взима от въведените в ресурсните полета стойности и инициализира
 // променливи от структурата на окръжността. След това вика: pDoc→AddBubble() ;
                Invalidate();
….
}
```

прерисуване на окръжности:

```
void CAutoBubView::OnDraw(CDC* pDC)
{
…
}
```

Дотук приложението е готово за компилация като самостоятелно приложение.

Процедура на добавяне на OLE Automation поддръжка към приложението - сървър

1. Class Wizard → Automation → Add Property с цел добавяне на нови даннови (get/set) променливи на automation обекта (CAutoBubDoc):

Implementation	тип	external name	Get функция	Set функция
Get/Set	long	Count	CountIs()	none

Функцията става метод на документния клас и следва да се опише.

Свойствата могат да бъдат не само get/set , а и прости член-променливи. Тогава даденото automation свойство става достъпно като даннов елемент на класа. Променливата ще бъде добавена в картата. В поле Notification function (ako сме избрали member-variable в диалог на развойната среда) може да се укаже функция, която се активира при присвояване от клиент на нова стойност на пропъртито. ClassWizard автоматично добавя записи в картата, декларации на функциите в класа и промени в ODL файла (както по-долу). За get/set – записът е DISP_PROPERTY_EX; за член променливи – DISP_PROPERTY.

2. добавяме и методи към класа. За всеки метод имаме: External Name – името на Automation метода и Intenal Name – името на съответната member функция. В полето Return Type може да се постави всеки Automation съвместим даннов тип. Параметрите на метода се указват в Parameter List. MFC се занимава с преобразуването им в VARIANTARG и обратно за да е коректно повикването на Invoke().

(За случая такива методи могат да са напр. Clear, Add. Те стават методи на документния клас и следва да се опишат функционално.)

За всеки добавен метод Class Wizard прави:

1 добавя декларация в .h файла на класа
2. реализира празна функция във файла
3. добавя ред в dispatch картата на класа
4. добавя метода и disp_id в ODL файла на проекта.

Код за тези ф-ии следва да се добави в CAutoBubDoc.

Файлът .odl се създава автоматично от Class Wizard:

```
// AutoBub.odl : type library source for AutoBub.exe
[ uuid(06A6EDEB-A2FD-11CF-9C7D-000000000000),
 version(1.0) ]
library AutoBub
{
		importlib("stdole32.tlb");
		//  Primary dispatch interface for CAutoBubDoc
		[ uuid(06A6EDEC-A2FD-11CF-9C7D-000000000000) ]
		dispinterface IAutoBub
		{
			properties:
				//{{AFX_ODL_PROP(CAutoBubDoc)
				[id(1)] long Count;
				//}}AFX_ODL_PROP
			methods:
				//{{AFX_ODL_METHOD(CAutoBubDoc)
[id(2)]			void Clear();
[id(3)]			void Add(long clr, short nYPos, short nXPos, short nRadius);
				//}}AFX_ODL_METHOD
		};
		//  Class information for CAutoBubDoc
		[ uuid(06A6EDEA-A2FD-11CF-9C7D-000000000000) ]
		coclass CAutoBubDoc
		{				[default] dispinterface IAutoBub;
		};
		//{{AFX_APPEND_ODL}}
};
```

Последната промяна в кода на Automation сървъра е с цел изобразяване при инстанцииране от страна на контролер.
Търсим CAutoBubApp::InitInstance().

там където се проверява дали приложението е стартирано като OLE server или като самостоятелно приложение се добавя код за регистриране на временен сървър , което позволява външни приложения да поискат поддръжка от OLE библиотеките за да създават OLE обекти от този тип в рамките на своето приложение. Ето част от кода:

```
BOOL CAutoBubApp::InitInstance()
{
…………..
// проверка дали е стартирано като OLE сървър
if(cmdInfom_bRunEmbedded || cmdInfo.m_bRunAutomated)
{           COleTemplateServer::RegisterAll();          // регистрира като running всички OLE
                                                         // factories за сървъри. Това позволява
                                                         // създаване на обекти от такъв тип
                                                         // от други приложения.

//  следва ваш код проверяващ дали има опция за /Automation при стартиране на
// програмата. Ако да – изобразява главния прозорец на този клас
if( !ProcessShellCommand(cmdInfo))
            return FALSE;
return TRUE;}   /* следва автоматично добавен код */……………..
}
```

С това OLE Automation сървъра е готов.

Създаване на Automation controller

1. На Visual C++

С MFC Automation клиент се пише лесно. COleDispatchDriver осигурява извличане на dispatch указатели и методи за достъп до методи и свойства.
Създават се производни на горния класове. С тях достъпът до automation методите и свойства е елементарен.

За целта в средата на ClassWizard избираме AddClass, после From a Type Library. Там указваме типовата библиотека за конкретния сървър. Генерира се клас съдържащ функции за извикване на методите на сървъра (със същите имена) и Get/Set функции за достъп до пропъртитата му.

Например ако сървърът е CAutoMath с Prog ID “Math.Object”, то в контейнера следва да има код от вида:

```
CAutoMath math;
math.CreateDispatch(_T(“Math.Object”));
// създава Automation обект и IDispatch указател в променливата m_lpDispatch
// оттук обръщения към методи и свойства автоматично се превеждат през IDispatch
// повиквания:
int sum = math.Sum(2,4);          // достъп до метод
double pi = math.GetPi();         // достъп до пропърти
```

2. На Visual Basic

Може да се създаде приложение, което управлява AutoBub през неговия OLE Automation интерфейс.

1. Създаваме си начален диалог за въвеждане на данни за окръжностите, които се предават към сървъра след активирането му.
2. Декларираме обект – носител на сървъра (напр. обект с име BubbleMachine):

Dim BubbleMachine As Object

3. Създава се инстанция на AutoBub сървъра:

set BubbleMachine = CreateObject("AutoBub.Document")

след края на използването на сървъра той се освобождава:

Set BubbleMachine = Nothing

4. Описва се процедура, активираща се например от VB меню-опция "Add Bubble", която приема стойностите въведени чрез диалога и ги подава на BubbleMachine:

```
Private Sub IDC_ADD_Click()
' чете въведените в полета стойности например за цвят, координати и радиус
' формира
ColorRef = RGB( x,y,z)
' и други
' създава новата окръжност:
BubbleMachine.Add ColorRef, коорд_x, коорд_y, радиус
                                        използва се функция от сървъра.
End Sub
```